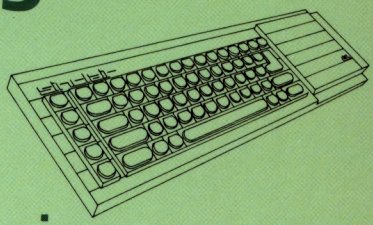
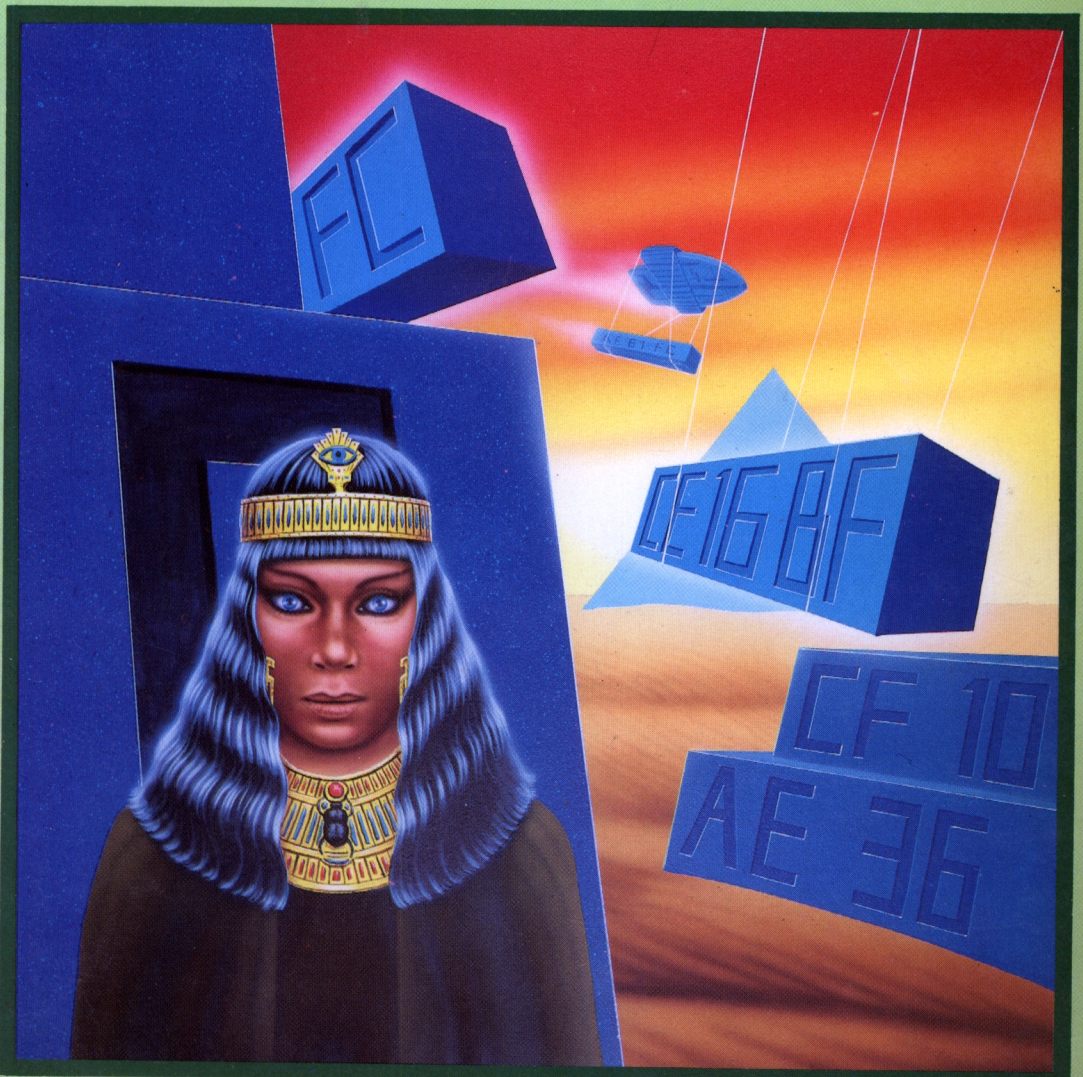
Assembly Language Programming on the Sinclair QL

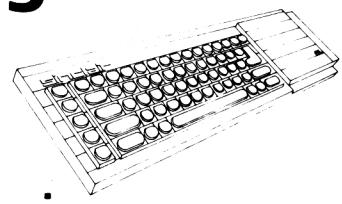


Programming the 68008 microprocessor

Andrew Pennell



Assembly Language Programming on the Sinclair QL



Programming the 68008 microprocessor

Andrew Pennell

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2R 3LD

Copyright © Andrew Pennell, 1984

℗ Sinclair QL, QL Microdrive and SuperBASIC are Trade Marks of Sinclair Research Ltd.
© The contents of the QL are the copyright of Sinclair Research Ltd.
℗ Quill, Archive, Abacus and Easel are Trade Marks of Psion Software Ltd.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

British Library Cataloguing in Publication Data
Pennell, Andrew
Assembly language programming on the Sinclair QL.
1. Sinclair QL (Computer)—Programming
2. Assembler language (Computer program language)
I. Title
001.64'24 QA76.8.S625

ISBN 0-946408-42-4

Cover design by Grad Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset by Paragon Photoset, Aylesbury.
Printed in England by Short Run Press Ltd, Exeter.

CONTENTS

		Page
Introduction		vii
1	Memory, Bits and Bytes	1
2	Inside the QL	7
3	The MOVE Instruction and Addressing Modes	13
4	Condition Codes, Branching and Arithmetic	25
5	Further Instructions and Passing Parameters	41
6	Exception Processing, Traps and Interrupts	47
7	Using the Hardware and Firmware	57
8	An A–Z of the 68008 Instruction Set	69
9	The 68008 Disassembler	133
10	Other 68000 Series Devices	155
Epilogue: Multi-tasking — an Example		161
References		165
Index		167

Contents in detail

CHAPTER 1
Memory, Bits and Bytes
Memory — ROM — RAM — hex — binary — two's-complementing — SuperBASIC functions.

CHAPTER 2
Inside the QL
What makes the QL tick — 68008 registers — status register — supervisor and user modes — memory map — storing machine code — hex loader.

CHAPTER 3
The MOVE Instruction and Addressing Modes
The MOVE instruction — addressing modes — stack pointer — other MOVEs.

CHAPTER 4
Condition Codes, Branching and Arithmetic
Condition codes — SUB instruction — branches — CMP instruction — looping using DBcc — subroutines — CLR instruction — ADD — OR — EOR — shifts and rotates — printing in hex.

CHAPTER 5
Further Instructions and Passing Parameters
LEA — NOP — EXG — NOT — NEG — SWAP — passing parameters.

CHAPTER 6
Exception Processing, Traps and Interrupts
Exception processing — vector table — interrupts — traps — user-defined exceptions.

CHAPTER 7
Using the Hardware and Firmware
The screen — colour modes — plotting points — printing characters — the 8049 — scanning the keyboard — the alternate screen — disabling QDOS.

CHAPTER 8
An A–Z of the 68008 Instruction Set
Calculating addressing bytes — addressing modes — complete instruction set.

CHAPTER 9
The 68008 Disassembler
The algorithm — disassembler — description — converting to machine code — full version.

CHAPTER 10
Other 68000 Series Devices
Other 68000 series processors — support devices — the 68881 maths processor — 68486/7 graphics system.

Introduction

The Sinclair QL has been hailed as a major breakthrough in personal computing and in part this is because of the microprocessor it uses — the Motorola 68008. Since its launch the machine has been the subject of much controversy particularly when delivery dates were not fulfilled. Even before its production problems were known there was comment on the claims made by the advertisements saying that the QL was a '32 bit' machine. Critics argued that it was only an 8 bit machine.

This book is concerned with the 68008 and programming it on the QL. I would hope that by the end of this book you will be able to form your own conclusions as to whether it is a 32 bit device or not. I personally feel that it has the power of a 32 bit chip, but not the speed.

If you have programmed any 8 bit devices before, you will probably find programming the 68008 easy, compared with whatever you've used previously. On the other hand, if this is your first processor, you will think all the 8 bit chips are positively prehistoric in comparison, and you could well be right!

Credit where it's due

I would like to express my sincere thanks to the following:

My parents and friends, for their patience while I wrote this;
Mike Salem and David Link for their invaluable technical assistance;
Sinclair Research for inventing the QL and supplying technical information;
Motorola for inventing the 68000 series and supplying data on it;
All those at Sunshine for persuading (and bullying) me to get this done;
and Apple, for teaching Macintosh about Man.

Andrew Pennell
Cliftonville, Kent
August 1984

CHAPTER 1
Memory, Bits and Bytes

This chapter aims to introduce all the technical terms used in the rest of the book, and is intended for the newcomer to machine-code programming. However, even if you've done machine code on other machines, you should still find some of this is new to you.

The two most important parts inside any computer are its memory, and its processor. The memory stores all the programs and data, which is required for the processor to work properly. Within the memory, everything is stored as numbers, even if it seems as if it is storing words, like LET and PRINT. What the numbers actually mean and do depends on how they are used. Memory is referred to by its *address*. A memory address is a number from 0 up to, well, it depends, normally on the QL it stops at 262143. At each address, *data* is stored, in the form of a number from 0 to 255, which is known as a *byte*. All memory can have its contents inspected, by *reading*, and in SuperBASIC the function PEEK is the way to read memory. For example, to see what numbers lie from address 0 up to address 40, a BASIC program like this could be used:

```
10  FOR i=0 TO 40
20  PRINT "At address ";i;"the data is";PEEK(i)
30  NEXT i
```

which will produce various numbers, and you will notice that all the data lies between 0 and 255, as it should. (Don't worry about what these particular numbers mean — they are explained later, in Chapter 7.)

There are two types of memory, RAM and ROM. ROM stands for Read Only Memory, and its contents are permanently fixed at the time of manufacture. It is used for storing things that must always be in the machine, and on the QL the ROM contains the SuperBASIC language, and the QDOS operating system.

RAM stands for Random Access Memory, and can be read, like ROM, and also written to, so its contents can be altered. Things like SuperBASIC cannot be stored in RAM because its contents are lost when the power is removed, unlike ROM. On the QL, the RAM is used for storing things that need to be changeable including the screen display and BASIC programs. To alter RAM contents from SuperBASIC, the POKE

command is used, followed by two numbers — the first is the address you want altered, and the second is the data you want to put in. POKing into ROM has no effect of course.

Hex and Binary

Most of us have ten fingers so we all count in *base 10*. It would be much more useful if we all had 16 fingers for computer work, as base 16, or *hexadecimal* as it is known, is a very common way of expressing numbers when programming in machine code. As we have only 10 digits, to denote numbers between 10 and 15, we use the first six letters of the alphabet. Thus, the number 12 decimal is C in hex. The way of denoting hex numbers varies on different machines, but on the QL the usual way is by preceding them with a dollar sign, so 12 is $C. With numbers greater than 15, we use extra digits, like we do with numbers greater than 9 in decimal, so the number 30 is $1E (as 30=1*16+14), and so on, for up to eight digits. There are two reasons for using hex. The first is that it is much more concise for bigger numbers — for example, the number 524288 is rather unwieldy compared with the hex version of $80000. The other reason is that it is easy to convert between hexadecimal and another base useful for computing — *binary*.

Binary

Binary is used frequently in computing and is number base 2. Thus you can have only the digits 0 and 1 in a binary number. For a byte, which ranges from 0 to 255 decimal, the binary equivalent ranges from 0 to 11111111. Each binary digit counts a power of two, so the rightmost digit is the 1s column, the next one the 2s, the next the 4s, and so on. So, to convert the decimal 30 into binary, we need to write it as a sum of powers of two, which works out to be 16+8+4+2, giving a binary of 11110. It is usual for binary digits to be written in multiples of eight, so 30 decimal is equivalent to 00011110 binary. Each binary digit is known as a *bit*, and a byte has eight bits. The bits themselves are numbered, starting at 0 for the rightmost, and going up. This can be made a little clearer by a diagram:

Bit number	7	6	5	4	3	2	1	0
Power of 2	128	64	32	16	8	4	2	1
Example:	0	0	0	1	1	1	1	0 = 30 decimal

The largest bit number is known as the *most significant bit*, which is bit 7 (byte), bit 15 (word), or bit 31 (long).

Half a byte is, believe it or not, a nibble, and has four bits. To convert between hex and binary, each nibble corresponds to a hex digit, so in the

example above the nibble 0001 corresponds to 1, and 1110 to E.

A byte is the basic unit of storage. 1024 bytes (2 ^ 10) is known as a kilobyte, denoted by K, and 1024K is known as a megabyte, or M. The 68008 uses two other units — *words*, and *long words*.

A word is two bytes, and must always start at an even numbered address. It can hold a value of between 0 and 65535, (=$0000 to $FFFF). The first byte in a word is known as the 'high byte', or most significant byte (MSB), and contains the leftmost hex pair of digits. The second byte is the 'lower byte', or least significant byte (LSB), and contains the rightmost pair of digits. For example, if the word data $1234 was stored at address 30, then location 30 would hold $12, and location 31 would hold the $34 part of the data. (This is the opposite order to most 8 bit processors.) SuperBASIC has two commands for handling word quantities — there is

```
PEEK_ W(address)
which is equivalent* to
256*PEEK(address)+PEEK(address+1)
```

for reading words from memory, and there is

```
POKE_W address,data
which is directly equivalent to
POKE address,data DIV 256: POKE address+1,data MOD 256
```

for writing words into memory.

A long word is four bytes (two words) and must also always be at an even numbered address. It can hold a value of between 0 and 4294967295 (=$00000000 to $FFFFFFFF). The first byte holds the most significant byte of the data (ie the leftmost two digits), then the next most significant byte, and so on for all four bytes. For example, if the long data $12345678 was stored at location 30, then location 30 would hold $12, location 31 $34, location 32 $56, and location 33 $78. There are two SuperBASIC commands for handling long quantities:

```
PEEK_L address
which is equivalent* to
65536*PEEK_W(address)+PEEK_W(address+2)
```

and

```
POKE_L address,data
which is equivalent to
POKE_W address,data DIV 65536: POKE_W address+2,data MOD 65536
```

(*The functions are not exactly equivalent because PEEK_W and PEEK_L give signed results. If the value is greater than $7FFF and $7FFFFFFF respectively then the number is two's-complemented — see below.)

Two's-complementing and sign extension

As we have seen, the different sizes of numbers can range from 0 to 255 (byte), 65535 (word), and 4294967295 (long). However, there is an alternative way of reading numbers, in order to get both negative and positive results.

We shall deal with byte quantities first. If a number is said to be held in 'two's-complemented' form, this means that the number can be positive or negative, depending on its highest bit — bit 7 in the case of bytes. If the bit is 0, or *reset*, then the number is as usual. However, if bit 7 is 1, or *set*, then the number is taken as negative, by subtracting 256 (in the case of bytes). Thus, numbers stored as 0 to 127 are the same if stored as two's-complemented, but numbers between 128 and 255 correspond to the values of −128 to −1 respectively (from 128–256 and 255–256). Thus in two's-complement form, a byte can have a range of −128 to 127 inclusive.

Two's-complementing with words is similar, except the bit that determines the sign is bit 15, and negative values are calculated by subtracting 65536. Thus numbers from 0 to 32767 mean just that, but numbers between 32768 and 65535 correspond to the values −32768 to −1, respectively. Thus a two's-complement word can have a value between −32768 to 32767.

With long words, the sign bit is bit 31, and negative values are calculated by subtracting 2147483648 (=$80000000), giving a range of −$80000000 to $7FFFFFFF inclusive.

Sign-extension is when the most significant bit of data is transferred to all the higher bits, when changing sizes. For example, the byte $86 would be sign-extended to a word of $FF86, and then to a long word of $FFFFFF86. Thus 'negative' numbers get FFs added, while positive numbers get 00s. The reason for sign-extension is to retain the same two's-complement value — $86 (byte) is −122, and so are $FF86 (word), and $FFFFFFF86 (long). The PEEK_W and PEEK_L functions always give signed results in SuperBASIC.

The processor

The processor in a machine is the controller of everything else. For it to work, it must have a machine-code program to execute, and programs are stored as numbers in the memory of the machine. It would be extremely

difficult to write machine code with just numbers, so designers of each processor invent a way of expressing what the processor does in combinations of words and numbers, called mnemonics. A machine-code instruction is written by us humans using mnemonics, then converted, either by hand or by another program called an *assembler*, into the numbers that the processor requires. The numbers corresponding to each instruction are known as *opcodes*. For example, the instruction

JMP $1234

while it probably doesn't mean much to you at the moment, means a lot more than its opcode would, which is $4EF81234. This book is concerned with programming the 68008 processor using mnemonics and instructions, and the actual method for converting instructions, that humans understand, into opcodes, that the processor understands, will be covered later.

When writing machine code, it's usual to denote all references to locations not by their address, such as $1234, but by a word, known as a *label*, like PRINT. This has the advantage of being easier to understand, and easier to work out. It's also common to add text to the ends of instructions, known as comments, that describe what the instruction does. For example, a line like

LOOP JSR PRINT print the character

is a lot clearer than the functionally equivalent

JSR $1234

In the first line both LOOP and PRINT are labels, with the comment following the first line after the instruction.

Useful SuperBASIC functions

Converting between decimal and hex can be a bit tricky at times, so why not use SuperBASIC to do the conversions. Functions *hex$* and *bhex$* take a word or long word respectively in decimal and convert it to a four or eight digit hex string, while function *dec* does the reverse.

Listing 1.1: Hex-Decimal Functions

```
100 DEFine FuNction h1$(a)
110 RETurn CHR$(48+a+7*(a>9))
120 END DEFine
```

```
130 DEFine FuNction h2$(a)
140 RETurn h1$(a DIV 16)&h1$(a MOD 16)
150 END DEFine
160 DEFine FuNction hex$(b)
170 LOCal a,h$
180 a=pos(b):IF a>32767 THEN a=a-32768
190 h$=h2$(a DIV 256)&h2$(a MOD 256)
200 IF pos(b)>32767 THEN h$(1)=h1$(h$(1)
+8)
210 RETurn h$
220 END DEFine
230 DEFine FuNction bhex$(a)
240 LOCal h1,h2
250 h1=INT(a/65536):h2=a-65536*h1
260 RETurn hex$(h1)&hex$(h2)
270 END DEFine
280 DEFine FuNction pos(a)
290 IF a<0 THEN RETurn 65536+a:ELSE RETu
rn a
300 END DEFine
310 DEFine FuNction dec(a$)
320 do_dec(a$):RETurn dd
330 END DEFine
340 DEFine PROCedure do_dec(a$)
350 LOCal s,t,q
360 t=0:q=LEN(a$)
370 s=CODE(a$(q))-48:IF s>22 THEN s=s-32
380 IF s>9 THEN s=s-7
390 t=t+s*16^(LEN(a$)-q)
400 q=q-1:IF q>0 THEN GO TO 370
410 dd=t
420 RETurn
430 END DEFine
```

(These functions are not the neatest way of converting between the bases. They have been written in such a way as to work on early bug-ridden versions of the QL, as more concise versions can crash such machines.)

CHAPTER 2
Inside the QL

Inside your QL is a lot of the latest in electronic technology. There are many components to it, but the main ones that interest us are shown in **Figure 2.1**.

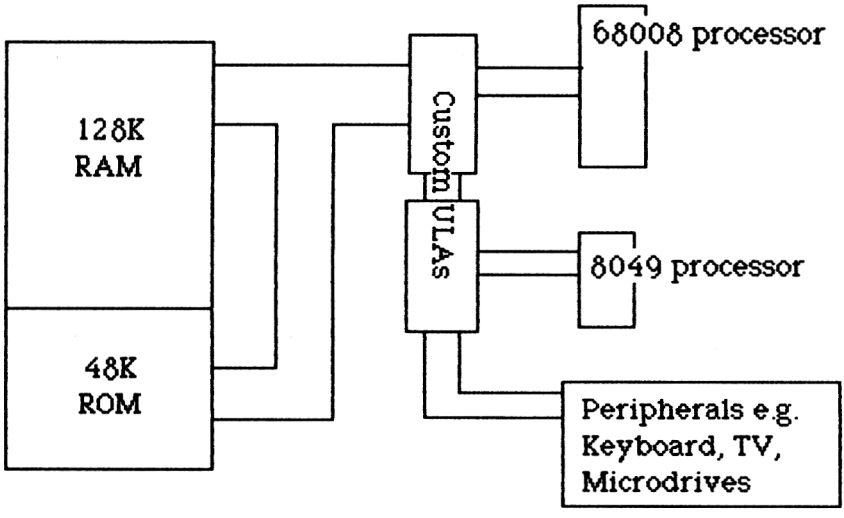


Figure 2.1: QL Main Components.

The main microprocessor is the Motorola MC68008, which works in conjunction with a slave processor, an Intel 8049. This book is mainly concerned with the former, though the use of the latter is covered in Chapter 7. The 68008 is a direct development from the MC68000, which was first introduced in 1979 in limited quantities. The -8 version is very similar to its predecessor, and is almost completely software-compatible. The main differences between them are the number of data and address lines — the 68000 has 16 data and 24 address lines, and can address 16 Mbytes, while the -8 has only 8 data and 20 address lines, and can address 1 Mbyte. The difference in data lines had the effect that the -8 runs rather slower than its counterpart. The 68008 is a very new device, first appear-

ing at the end of 1983. The Sinclair QL is the first computer to use the processor.

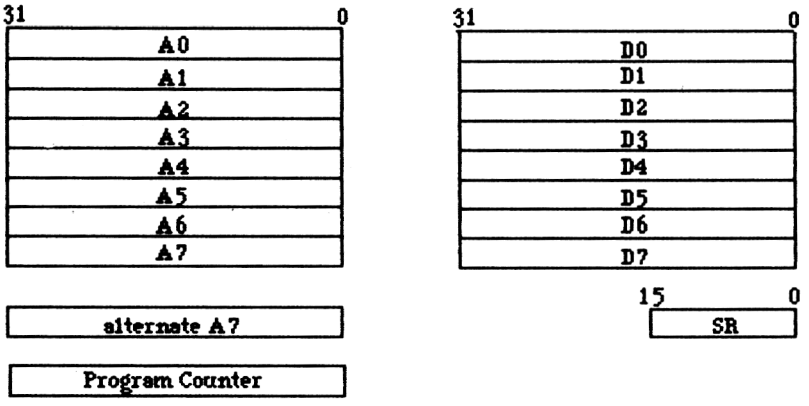


Figure 2.2: 68008 Register Set.

A simplified functional diagram of the 68008 is shown in **Figure 2.2**. Internally there are eight address registers, named from A0 to A7, a special alternate A7 register, eight data registers, named D0 to D7, program counter, and status register. All but the latter are 32 bit registers, while the status register (known as the SR) is 16 bit. Address register A7 is a special case, as it is also the stack pointer register. The 68008 can run in two different states, *supervisor* and *user* modes. Each mode requires its own stack pointer, hence the existence of two A7s. The current mode, and A7 usage, is defined by a bit in the status register. The uses of all the bits is shown in **Figure 2.3**.

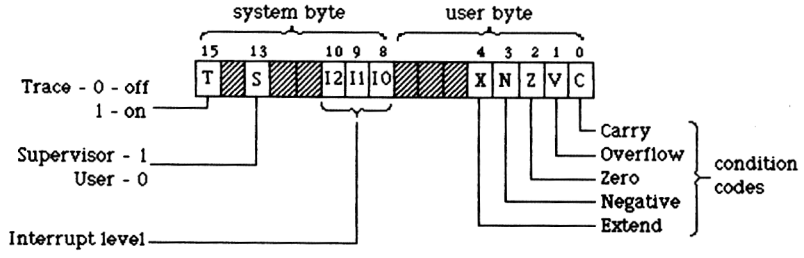


Figure 2.3: Status Register Bit Usage.

The high byte of the status register stores important system information, and can only be altered when in supervisor mode. Bits control *trace* mode, which allows single stepping among other things, supervisor or user mode, and the current level of interrupts. Interrupts and using trace are covered later in Chapter 6.

The lower byte of the status register (known as the CCR, for Condition Control Register) contains all the condition code bits, altered using arithmetic instructions, and tested for conditional jumps. Further details are given in Chapter 4. Note that there are six unused bits in the status register — these are never altered by any instructions, so you could use them for your own purposes. However, beware of getting into bad habits — other processors in the 68000 series do use these extra bits, and this applies to other non-standard techniques.

Supervisor and User modes

When the 68008 starts up, it is in supervisor mode. What happens subsequently depends on the software that executes, and in the case of the QL the machine soon goes into user mode. Generally speaking, the QL is in user mode, though it can be persuaded to temporarily go into supervisor. When in supervisor, the processor can really do what it likes. It has access to all the memory, and the whole status register, so it can go into trace mode, and change interrupt priorities. When in user mode, the machine is restricted. The upper byte of the status register cannot be altered, though it can be read, and certain other instructions are privileged. This means that they cannot be executed when in user mode, and if you try, well, various things can happen. Basically, an exception occurs, which can trap such instructions, but I won't go too far into this at this moment.

QL memory map

The standard QL has 128K of RAM and 48K ROM, with various other devices in its memory map. With reference to **Figure 2.4**, it can be seen that there are several areas that are defined but currently unused, and that out of the QL's complete address range of 1 Mbyte, there are no areas that are not defined.

Storing machine code

With such a large amount of RAM available, you'd think that there would be loads of places in which you can store your own machine code. However, if you just put it in an area that seems to be free, sooner or later the system will use this area, wiping out your precious code. To tell the machine that you want to reserve some memory, you must use the RESPR function. RESPR stands for 'reserve procedure space', and should be followed by a number in brackets. This number is the number of bytes you wish to be made available. Unfortunately, RESPR can work

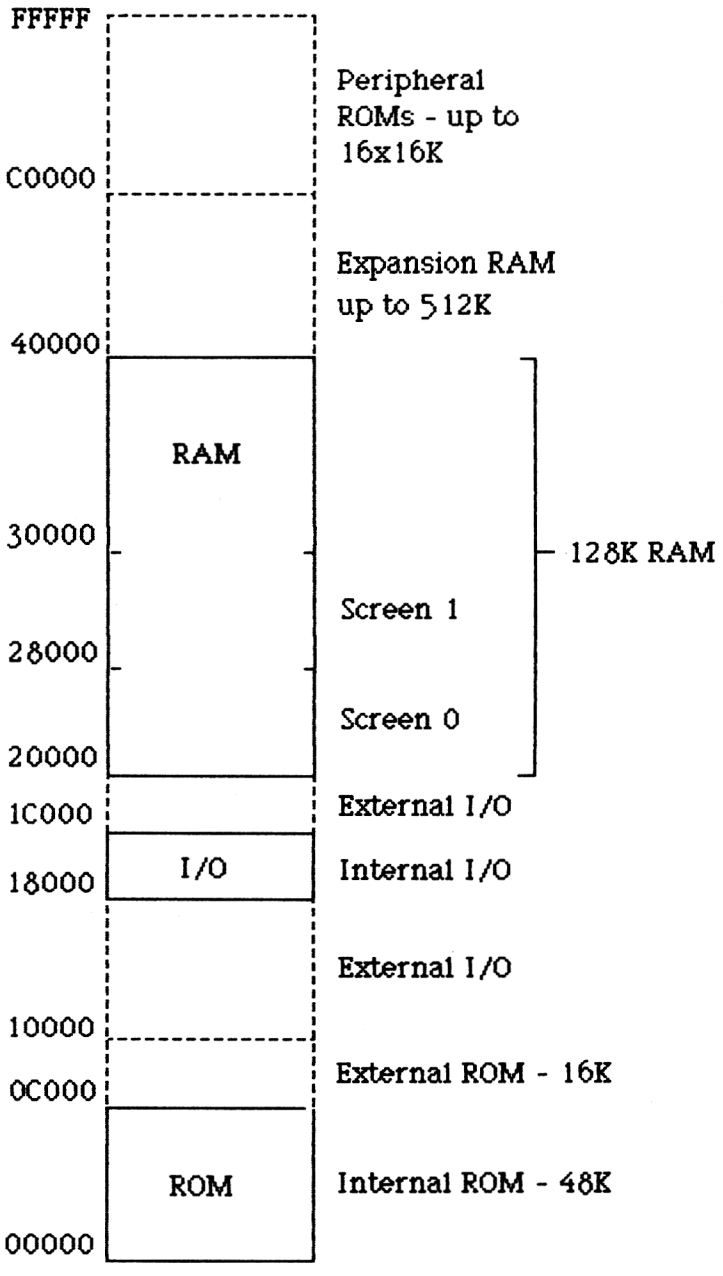


Figure 2.4: QL Physical Memory Map (not to scale).

differently on different machines, so for the moment we shall reserve space for all the routines in this book with the statement

a=RESPR(1024)

If you do this, regardless of which QL you have, 1024 bytes of memory will now be set aside for you. To find where they are, simply

PRINT a

which should be 261120 on a 128K RAM machine. A useful feature is that if you have a parameter of 0 in RESPR, it returns the first free location. Thus, after reserving space, RESPR(0) always tells you where your machine code may go. Unlike most other machines, on the QL there is no way of deciding first where you want your machine code to go — it will go where the system thinks it should. This means that any code you write must either be position independent, or have a special relocator. On older microprocessors, writing position-independent code is a positive nightmare, but on the 68008 it is no different to writing normal code.

Now we have a way of reserving machine code, we need a way of entering it into the machine's RAM. As the QL has no machine-code monitor, we shall have to write our own, in the form of a hex loader. **Listing 2.1** below is such a program, that requires DATA statements containing hex strings to be added from line 1000 onwards. The length of the strings does not matter, so long as they are even (ie complete bytes). Line 9999 must remain, as it tells the program when there is no more data.

Listing 2.1: Hex Loader

```
10 DEFine FuNction dc(a$)
20 IF CODE(a$)>=CODE("a") THEN a$=CHR$(C
ODE(a$)-32)
30 RETurn CODE(a$)-48-7*(a$>"9")
40 END DEFine
90 a$=""
100 INPUT "Start address? ";s
105 IF s<RESPR(0) OR s>=PEEK_L(163872) T
HEN PRINT "WARNING:not in proper RAM":ST
OP
110 RESTORE 1000:READ a$
120 IF a$="" THEN GO TO 250
130 a=dc(a$(1)):IF a<0 OR a>15 THEN GO T
O 200
```

```
140 b=dc(a$(2)):IF b<0 OR b>15 THEN GO T
O 200
150 PRINT !a$(1 TO 2);:POKE s,a*16+b
160 s=s+1
170 IF LEN(a$)<3 THEN READ a$:GO TO 120
180 a$=a$(3 TO LEN(a$))
190 GO TO 120
200 PRINT \"Data error at location ";s
210 STOP
250 PRINT \"Last byte loaded at ";s
260 STOP
9999 DATA "":REMark end marker
```

Firstly you are asked for a starting address, which is checked to make sure that it points to RAM above RESPR (PEEK_L(163872) calculates the top of RAM+1).

When entering machine code with this loader, either programs from this book or your own, there is an easy way to be sure that they *don't* work — by forgetting to reserve any memory, so RESPR(0) points not to free RAM, but to non-existent memory. Then, after you think you've POKEd your program in memory and try to execute it, the machine is very likely to fall over, losing your program.

We've now seen what is in the QL, where we can put our programs, and how we can enter them. In the next chapter, we shall start programming.

CHAPTER 3

The MOVE Instruction and Addressing Modes

The most often used instruction in the 68008's set is MOVE. It is a general 'load' instruction, and has two parameters — the source, and the destination. What it does is to take the source, whatever this is, and move it into the destination. This may sound horrendously complicated at first, but it isn't. Think of it as a LET statement in BASIC. The statement LET D0=0 has its equivalent of MOVE #0,D0. What it means exactly is 'Move 0 to register D0'. The hash is an important part of the instruction, as we shall see later. Our MOVE instruction above is not quite complete, as with all MOVEs you must specify the size of the instruction — whether you are MOVing a byte, a word, or a long word. You distinguish between them by following the MOVE with a full stop, then B, W or L, respectively. So if we want all of register D0 to be zero, the complete instruction is

```
MOVE.L #0,D0
```

Of course 0 is not the only number that can be moved to D0 — any number from 0 up to $FF (for byte operations), or $FFFF (for word), or even $FFFFFFFF (for long word operations). There are literally thousands of different types of MOVEs, but once you know a few, the others follow — honestly! For example, a similar instruction to that above is

```
MOVE.L #0,A0
```

which puts zero in register A0, in exactly the same way as that above. There is a slight restriction when moving data to an address register — only word and long word sized moves are allowed — byte-sized is not. In normal programming, this is not really a problem, as byte-sized moves are seldom required.

The # in the instructions above means 'immediate addressing': think of it as 'immediately putting the data' into the register. Immediate addressing is the simplest of the twelve different addressing modes — this may

seem a lot to those of you with experience of 8 bit chips, and is one of the main reasons behind the power of the 68000 series.

Direct addressing

The next addressing mode we shall look at is direct addressing. It's a nice, simple one, and is used for transferring data directly from a register. For example,

```
MOVE.L D3,D1
```

transfers whatever was in D3 into D1 (think of the comma as TO). D3 itself is not altered, so its BASIC equivalent would be LET D1=D3. Direct addressing can also be used with address registers, so 'MOVE.L A3,A0' is valid, and so is 'MOVE.L A3,D0' and 'MOVE.L D0,A3' The latter two show how you can transfer between registers regardless of type, though once again you cannot MOVE byte-sizes to or from an address register directly, so 'MOVE.B D3,A0' is not valid.

Indirect addressing

This addressing mode can only be used with address registers, and acts like a PEEK in BASIC. For example,

```
MOVE.L (A0),D0
```

means 'move the data from location A0 to register D0', or in BASIC, LET D0=PEEK_L(A0)

The brackets around A0 distinguish the mode from the direct mode. To compare the two, let's have a look at two apparently similar programs, and what each one does:

```
MOVE.L #$28000,A0        MOVE.L #$28000,A0
MOVE.L A0,D0             MOVE.L (A0),D0
```

The first instruction is the same in both programs, and puts the value of 28000 hex into register A0. The next lines look similar, but the important difference is the brackets on the one on the right. On the left, direct addressing is used to transfer the contents of A0, which we know to be $28000, into reigster D0. On the right, it's *the contents* of location $28000 that gets put in register D0. I hope you can now see the difference between direct and indirect addressing — if not, have another read.

Post-increment addressing

This one is a natural progression from indirect addressing. It simply increments the address register after the read, by a value depending on the side of the operation. For example,

MOVE.L (A0)+,D0

does the following: it reads the long word at location A0, puts it into D0, then increments A0. As this is a long operation, A0 gets 4 added to it. If this were a word or byte operation, it would get incremented by 2 or 1 respectively. The mode is shown by putting the address register within brackets, followed by a plus sign. It is extremely useful in many applications, including searching, filling and moving. The only 8 bit processors with this ability are also from Motorola, including the 6809.

Pre-decrement addressing

As you may gather from the name, this is the opposite to post-increment addressing. It is similar to indirect addressing, but the register gets decremented by a value before the memory read. So,

MOVE.L -(A0),D0

would decrease A0 by four (as this is long), then read location A0 and put it in D0. As with its partner, word and byte operations produce decrements by 2 and 1, respectively. It is important to remember that the register gets decreased *before* the read, and not after as with post-increment. The clue to which one does what is in their names. The mnemonics used may help too — the minus sign comes *before* the brackets around the register.

Indirect with displacement

OK, I know it's a mouthful, but you don't have to remember these official titles — as long as you remember the mnemonics, and what they do, the names aren't that important. Anyway, this is another variation on indirect addressing, this time with a number tagged on. For example,

MOVE.L 4(A0),D0

looks at location A0+4, and places its contents in D0. The number that precedes the first bracket is the displacement, and gets added to the value of A0 before the memory is read. Note that it doesn't actually alter A0, the result of the addition being stored temporarily. The value of the

displacement can take up to 16 bits, ie from 0–65535 (or 0–$FFFF). However, it is important to note that the displacement is taken as sign-extended, which means that 0–32767 means just that, while 32768–65535 means –32768 to −1, respectively. That is, if the displacement is greater than 32767, then you have to subtract 65536 from it. This allows negative displacements, so that

```
MOVE.L 65534(A0),D0
```

is equivalent to

```
MOVE.L −2(A0),D0
```

which has a BASIC equivalent of LET D0=PEEK_L(A0−2). This may seem quite complicated at first, but you should soon see the principle. If you don't, have another read, because the next one is a bit trickier!

Indirect with index

Most of the more complicated addressing modes are developments of simpler ones, and this is no exception. It is basically like the previous mode, but with the addition (literally) of another register. So, the instruction

```
MOVE.L 8(A0,D1),D0
```

does rather a lot. Firstly, it adds A0, D1 and the displacement — 8 in this case — all together, and stores the result temporarily. Then, it looks at that memory location, reads what's there, and puts it in D0 — wow! It's not quite that simple, though, as the displacement may only be eight bits, again sign-extended, giving a range of −128 to 127. In addition, the index register, D0 in this case, can be taken as word or long in size, and is sign-extended appropriately. Although I have used a data register as the index above, you can also use an address register, though in both cases you should specify its size, by following it with a .W or .L to suit. Thus,

```
MOVE.L 4(A0,A2.W),D0
```

has a near BASIC equivalent of LET D0=PEEK_L(A0+A2+4). I say 'near' because it doesn't take account of any sign-extending on the displacement or index register.

Having covered nearly all the most complex modes, let's have a look at a couple of simpler ones — the absolute addressing modes.

Absolute addressing

There are two modes of absolute addressing — short and long. Short addressing uses a word as an address, while long addressing uses a long word. The instruction

```
MOVE.L $18000,D0
```

is a long absolute address, as $18000 must be a long word. What the instruction does is simply to read the contents of location 18000 hex, and place them in D0. In other words, it is equivalent to the BASIC LET D0=PEEK_L($18000).

The short form is similar, but the address has to be a word, which is sign-extended so that 0–$7FFF are as you would expect, but $8000–$FFFF refer to addresses $F8000 to $FFFFF, respectively. Unfortunately, the advantages of short addressing on the 68000 series are not really relevant to the QL, as it is very seldom indeed that you would want to examine either of these areas in this way. This is because the QL memory map does not really follow the norm for 68000 series machines. The short addressing modes were intended to access ROM routines at the bottom of the map and I/O devices at the top, but the latter are not there on the QL, while it is bad practice to do the former.

Program counter addressing modes

A very neat feature of the 68008 is its ability to run position independent code very easily. This is inherently difficult on most 8 bit chips, with the exception again being the Motorola family.

The 68008 accomplishes this by having two special addressing modes, similar to those mentioned previously, but differing by being totally relocatable. This is done by storing not absolute addresses in the instruction, but just the displacement from the current instruction to the desired address. Thus it is possible to write programs without knowing where in the memory map they will go, as they will function anywhere. Both PC modes allow a 16 bit sign-extended displacement, allowing references forward $7FFF bytes, and backwards $8000 bytes — this is enough for most software. Indeed, most of the 48K ROM in the QL is totally position independent, apparently for speed and memory reasons.

There are disadvantages to PC modes though — they take longer to work out if you are hand assembling, and there are also limits on when they can be used, sometimes forcing you to use an extra instruction to do an intermediate step.

Program counter with displacement

This is the simplest, and is similar to absolute long addressing, but is position independent. It is denoted in the mnemonic by following the address with (PC), thus

```
MOVE.L LABEL(PC),D0
```

is functionally equivalent to

```
MOVE.L LABEL,D0
```

but position independent. It will only be allowed if LABEL is not further than 32767 bytes away from the instruction.

Program counter with index

This is vaguely similar to indirect with index, but with less parameters. For example,

```
MOVE.L TABLE(PC,D1.W),D0
```

will read the memory location TABLE+D1, and place the result in D0. As is usual for index registers, D1 in this case is sign-extended, and could be replaced by an address register, and be either word or long word in size.

Source and destination

In all of the above examples, the address mode under examination has been the source in the MOVE instruction, while the destination has been either a data or address register. This has been done for simplicity only, as the 68008 allows a large amount of mixing of the modes, and most of those above can also be used as destinations in MOVEs. The difference is that, instead of memory being read, it is written to. This is best illustrated by example:

Mode	**Example**	**BASIC version**
Indirect	MOVE.L D0,(A0)	POKE_L(A0),D0
Post increment	MOVE.L D0,(A0)+	POKE_L(A0),D0: A0=A0+4
Pre decrement	MOVE.L D0,-(A0)	A0=A0-4: POKE_L(A0),D0
Indirect with displacement	MOVE.L D0,8(A0)	POKE_L(A0+8),D0
Indirect with index	MOVE.L D0,2(A0, D1.W),D0	POKE_L(A0+D1+2),D0
Absolute	MOVE.L D0,$28000	POKE_L($28000),D0

As you may notice, there are a few missing from the above — firstly, immediate addressing. That's because it doesn't make sense to try something like MOVE.L A0,#$28000 as you can't move anything to immediate data. The others missing are the two PC addressing modes, and they are missing because the 68008 hardware invisibly distinguishes between data and program, and it doesn't like to allow the alteration of program area. This is an unfortunate restriction, and is one of the reasons why extra instructions are sometimes needed when position independent code is being written.

Up to now, all the MOVEs covered have been to or from a register, and users of other processors may not be surprised by this. However, an important feature of MOVE is its ability to transfer data directly from memory to memory, without having to go via any other registers. For example, suppose you want to copy the word data from the location pointed to by A0, into the location A2+D1+6. One way of doing it is like this:

```
MOVE.W (A0),D0
MOVE.W D0,6(A2,D1.L)
```

It uses register D0 as a temporary place to store the result, and works perfectly well. However, if you want a bit more speed, and you don't want to use another register, you can combine the two by doing

```
MOVE.W (A0),6(A2,D1.L)
```

In fact, you can have any addressing mode as the source, and most addressing modes as destination, with any combination you like. The limits described above regarding destination addresses still apply though.

It is important to remember that all word and long word memory references are to *even* numbered addresses — if they are not, dire things will happen. (Later we shall see how to detect such an occurrence.) Also worth remembering is that an address register cannot directly be used as a source or destination for a byte-sized transfer.

Word and byte-size accesses

All of the examples so far have been of long-sized MOVEs — you may be wondering what happens if you use word or byte-sizes. Well, the answer is, it depends on the destination of the MOVE. If the destination is an address register, then word-sized data will get sign-extended to 32 bits before going into the register. If it is a data register, only the lower 8 or 16 bits will be affected by the instruction, depending if it is byte or word. The

remaining bits will remain unchanged. If the destination is a memory location, then again only the appropriate number of bits will be affected.

The first program

If you're itching to try some 68008 code, here is the first listing. Before explaining it, it's worth pointing out a few points about machine code on the QL. It is executed with the CALL command, followed by the address of the routine. You should be careful with a couple of the registers — in particular, leave A6 alone, as it is an important pointer for the system, and A7, the user stack pointer, should always be back at its original value when you have finished. To leave your machine code and get back to BASIC, the RTS instruction is used. This stands for 'Return from subroutine', and is more fully explained in a short while. Before doing the RTS though, you should always zero register D0. If you do not, you will get an error message on return to BASIC, the message depending on the value of D0.

Anyway, back to the program. It's not amazingly impressive, as it doesn't do very much, but it does serve its purpose — it's the first machine code you have done, and here it is:

Listing 3.1: First Program

```
4280                CLR.L   D0      zero D0
4E75                RTS             and back to BASIC
```

To enter it, type in the hex pairs shown down the lefthand side, using the hex loader, and to test it try CALL RESPR(0). If all is well, you should simply get the cursor back. If not, you've managed somehow to get a four-byte program wrong!

Now is probably the best time to explain a few things about opcodes in general. It is imperative that every instruction starts at an even-numbered address. Be particularly wary of this if your program includes blocks of data in byte chunks. In addition, every instruction is an even number of bytes in length, with a minimum of two, and a maximum of ten. You will seldom find your own programs needing instructions of over six bytes in length though.

You may well be wondering how I worked out the hex for the program. There are two ways of doing it — the easy way, and the hard way. The hard way is to do it by hand, by using the codes in Chapter 8. It's not too difficult for small programs, but alterations can be difficult and it is prone to error. The easy way is to use an assembler, though at the time of writing none is available for the QL. I must admit, most of the programs in this

book I did using an assembler, though not on the QL itself — it was done on another machine, and the bytes downloaded via the RS232 port. I didn't have the assembler originally, though, so I am used to hand-coding, and I sympathise with those of you without an assembler. Even so, you can learn an awful lot more by doing it the hard way. If you've done it for other processors, and thought it was quite easy, you will have a shock when you start on the 68008, particularly regarding the number of bytes used. However, as with most things, you can soon get used to it, and your speed will usually increase.

The stack pointer — A7

At this point it's worth having a look at the way the 68008 stack works. In fact, there are two stacks, one used when in user mode, and the other when in supervisor mode. The stack pointer is address register A7, which means that there are actually two register A7s in the processor. When you're in user mode, which is most of the time, you can only get to the user type of A7. If you're in supervisor, though, you can access both of them. For the moment, let's stick to user mode.

Where is the user stack on the QL? Well, it depends, but after a CALL it is somewhere high in memory, just below RESPR(0). As long as you don't have your 128K machine totally full, you should never run out of stack space. Like many other processor's stacks, the 68008's is 'upside down'. This means that the bottom of the stack is at a higher address than the top! Thus, when you put something on the stack, A7 decreases, and when you remove something, it increases. The value of A7 always points to the last item on the stack (always the most significant byte) which makes it very easy to manipulate it using pre-decrement and post-increment addressing. For example, if in a program you want to save the value of D1 on the stack before you do an operation, and afterwards you want to get it back, you can do this by:

```
MOVE.L D1,-(A7)      puts it on the stack
.....                do whatever
.....
MOVE.L (A7)+,D1      then get it back
```

The main use for the stack is holding return addresses of subroutines, but we'll cover this later. Finally, a word of warning — it's best not to try a byte-sized operation on the stack. Although it's perfectly legal to do so (so long as it doesn't refer directly to an address register) the results will not be as expected, as it will be converted to a word operation. If the value of A7 were to become odd, it would be very dangerous, as if the system

tries to do a word or long word access subsequently using A7, such as a subroutine call or interrupt, the processor will literally stop, and your program will be lost. Therefore, as a golden rule — never do byte operations on the stack pointer.

Other sorts of MOVEs

There are a few other types of MOVE instruction, with widely varying uses. There are MOVEs for the condition codes, status register, user stack pointer, and the specialist instructions MOVEM, MOVEP, MOVEQ and MOVEA. I shall now cover some of these, but the remainder I shall leave for Chapter 8.

MOVEA stands for Move Address, and is the same as a normal MOVE, but with the destination being an address register. It is officially separate from the MOVE instruction, but it only confuses the matter, so you can safely ignore it. All 68000 assemblers seen so far don't require the extra A, and I don't expect any to.

MOVEQ stands for Move Quick, and is a faster (and less memory-consuming) way of doing MOVE.L #??,D0. The immediate data can be up to eight bits in length, sign-extended to a full 32 bits, giving a range of 0–127 and $FFFFFF80 to $FFFFFFFF. For example

```
MOVEQ#12,D0
```

puts the long value of 12 into register D0. Note that a size isn't needed, as it is always taken to be long, and that there is no quick instruction for address registers.

MOVEM stands for Move Multiple, and is a fast and easy way of putting any number of registers into or out of memory. It is particularly useful for saving and restoring registers on the stack, and this is undoubtedly the most popular use of the instruction. For example, to save the registers D0 to D4,A0 to A3 and A6 on the stack the instruction would be

```
MOVEM.L D0-4/A0-3/A6,-(A7)
```

Note how the register list is specified — consecutive registers are separated by dashes, while others are separated with slashes. To save items on the stack, A7 is used in pre-decrement mode, but many other addressing modes can be used as the destination. For exact details on which ones are allowed, see Chapter 8.

To restore registers from the stack, post-increment addressing with A7 is used, and to restore the registers above the instruction would be

MOVEM.L (A7)+,D0–4/A0–3/A6

The order of the registers in the list is not important, as the actual order in which they are stacked and unstacked is determined by the processor.

MOVE with status register

To transfer the 16 bit status register to or from another register or memory, simply put SR in either the source or destination. It is always a word instruction, due to the size of the SR. Beware though — MOVing anything TO the SR can only be done when in supervisor mode.

The 8 bit condition code register (ie the low byte of the SR) can also be specified in either the source or destination using CCR, in either user or supervisor mode. Despite its size, all such MOVEs are word-sized too, but only the lower byte is actually used.

CHAPTER 4

Condition Codes, Branching and Arithmetic

Although MOVing data about is useful, it's not really much good on its own. It is a bit like programming in SuperBASIC using just LET, without GOTO, GOSUB, IFs and DEFs, and without any maths functions at all. In this chapter we'll cover the condition codes, branches, jumps, and subroutines. Towards the end, we'll also come across the various arithmetic operations.

The condition codes

The condition codes are stored in the lowest five bits of the status register, and cover five types of condition, namely X (extend), N (negative), Z (zero), V (overflow), and C (carry). I'll deal with these one by one, but not in that order. Firstly let's look at an instruction that is one of the most common that alters them — subtract.

The SUB instruction

This stands for subtract, and is used for subtraction involving data registers. There are several forms, but let's start by looking at the instruction

```
SUB.L#8,D0
```

Notice that, as with MOVE, a size has to be specified, and in this case it is long. What this means is 'subtract 8 from the contents of D0', ie do the calculation D0=D0−8. The instruction doesn't just do the subtraction, but it looks at the result and alters the condition codes suitably. There are five condition codes, and we'll deal with each of them in turn, starting with the easiest.

Z — Zero flag

This is the simplest condition, and is self-explanatory. If the result is zero, then this condition flag is set, else it is reset. Thus, in the above example, the Z flag will be set only if D0 equalled 8 originally — if it equalled anything else, it will be reset.

C — Carry flag
The carry flag is set if the subtraction 'carries through zero'. Thus, if D0 was 7, and had 8 subtracted from it, the result would be −1, and the carry generated because the result does 'carry through zero'. If it was between 8 and 0 inclusive, no carry would be generated.

N — Negative flag
This flag is set if the result is 'negative'. As we are talking about two's-complemented numbers, negative means if the most significant bit is set. So for long operations, a result is negative if bit 31 is set, else it is positive. For word results, bit 15 determines its sign, and byte results use bit 7 as the sign bit. Thus in the above example, the N bit will be set if the result has bit 31 set, ie lies in the range $FFFFFFFF to $80000000 inclusive. So, if D0 was 10, the result of the subtraction would be −2 which is $FFFFFFFE, which is negative, so N would be set.

V — Overflow flag
The overflow flag is set after an operation if the result is 'too big' for the specified size, and will be set if the sign changes after a calculation when it shouldn't. For example, if you did SUB.B #8,D0 and D0 was originally −125, the proper answer would be −133, but this is too big for 8 bits, and the result would end up as 123, which is positive. As the sign changed when it shouldn't have, an overflow would be generated.

X — Extend flag
This is effected by certain operations only, and is set the same as the carry flag. It is a 'special' carry flag, used in multi-precision operations, such as floating point arithmetic. It is set to the same state as the carry flag by a Subtract.

The previous example used a long-sized subtract — if word or byte sizes are used, only the relevant part of the operands is read and altered so

```
SUB.B #9,D0
```

would set the Z flag if D0=9, or if the least significant byte is D0, so it would also be set if D0=$1009, or $F1234E09 for that matter.

Branches

No, this has nothing to do with trees. A branch is a type of jump, like a GOTO in BASIC. There are no less than 15 different branch instructions, referred to generally as Bcc (for Branch on Condition Code). The simplest is BRA, which stands for Branch Always, and should be

followed by a program label. When executed, it means 'transfers control to whichever location the label is', so an instruction

BRA START

would cause a jump to whatever location START is. There is a restriction on all branches though — they can only be to locations within 32767 bytes of the instruction. There are eight other straightforward branch instructions, namely

BEQ	branch if equal (Z)
BNE	branch if not equal (not Z)
BCS	branch if carry set (C)
BCC	branch if carry clear (not C)
BMI	branch if minus (N)
BPL	branch if plus (not N)
BVS	branch if overflow set (V)
BVC	branch if overflow clear (not V)

As can be seen by the mnemonics, these eight are directly related to the four condition codes. Note that there are no conditional branches on the extend flag. There are some more types of branches, but they are not really applicable to SUB, only CMP, which we'll see later.

There is another way of branching — using the JMP instruction, which stands for Jump. It can be followed not just by an address, like Bcc, but by several different addressing modes, though its use in your own programs is limited on the QL as it is not normally position independent.

Other types of SUB

The usual SUB instruction has one of two general forms — either

SUB (address),Dx

or

SUB Dx, (address)

The first type includes the above examples, and subtracts the contents of (address) from the data register. The second type does a similar subtraction, but the other way round, in other words it subtracts the value of the data register from the contents of the address, then puts the result back in the address. To help remember which is which, think of the

comma as 'from'. Here are some examples, with their BASIC equivalents:

```
SUB.B (A1),D1          D1=D1-PEEK(A1)
SUB.L $28000,D4        D4=D4-PEEK_L($28000)
SUB.W D4,6(A1)         POKE_W A1+6, PEEK_W(A1+6)-D4
```

Another type of subtraction is SUBA, for subtracting values from address registers. It has the general form

SUBA (address),Ax

This cannot be byte-sized, and there is no equivalent for SUB Ax,(address), though. There are other differences too. The condition codes are not altered a scrap by SUBA and, with a word-sized SUBA, both parameters are sign-extended to 32 bits. Another type of subtraction is SUBI, for Subtract Immediate, and it has the general form

SUBI #(data),(address)

It subtracts the immediate data from the contents of the address, as you would expect. The final form of SUB is SUBQ, for Subtract Quick, of the form

SUBQ #(data),(address)

It is similar to SUBI, but very much faster and less memory-consuming. The data can only be from 1 to 8 inclusive, though. The way an instruction, like SUB, can have different forms is repeated for several other instructions, and a good assembler should automatically distinguish between them.

The CMP instruction

This stands for 'compare', and is used, not surprisingly, for comparing things with data registers. It has the general form

CMP (address),Dn

where (address) can be any of the modes described in the previous chapter, without restriction. There are many different types of CMP and,

like MOVE, it has to have a size — byte, word or long, denoted by the usual .B, .W or .L following the CMP. For example, the instruction

CMP.L #8,D0

'compares' 8 with D0. What the processor does is to subtract the (address) from the data register, but not store the result anywhere. Although the result is not stored, the values of the condition code register are affected, depending on the result of the subtraction. So, with the above instruction, the processor does the subtraction

'contents of D0'−8

Other compare instructions

The normal CMP instruction compares a parameter with a data register. There is a similar instruction CMPA, which compares a parameter with an address register. Note that CMPA cannot have byte-sized operations, and that word-sized CMPAs are sign-extended to 32 bits before the operation.

There is another compare instruction, namely CMPI, which stands for 'compare immediate'. It has the general form

CMPI #data,(address)

and can be byte, word or long in size. The size of the immediate data is always the same as the size of the instruction, (address) can be most of the different addressing modes, with the exceptions of address register direct, both PC modes, and immediate mode. Thus these are invalid:

CMPI.L #5,A0	can be replaced by CMPA.L #5,A0
CMPI.L #7,TYPE(PC)	
CMPI.L #9,#6	doesn't make sense!

More branching

As well as the conditions described previously, there are some additional ones usually only useful after a CMP instruction. They do not act directly on each condition code bit, but use combinations to produce useful extra conditions. Firstly, there are two that use combinations of the carry and zero flag, namely

BHI	branch if higher
BLS	branch if lower or same

So, after the instruction

CMP.B #20,D0

a BHI would be executed if D0 was HIgher than 20, ie 21 to 255 inclusive, and a BLS executed if it was Lower or the Same, ie 0 to 20 inclusive.

As well as these conditions, there are six others, which work on signed values:

BGT	branch if greater than
BGE	branch if greater than or equal to
BLT	branch if less than
BLE	branch if less than or equal to

These do not use single bits of the condition code register, or even pairs, but combinations of the N, V and Z flags, to produce these extra conditions. Don't confuse 'less than' with 'lower' — the former takes into account the signs of the operands, while the latter takes absolute values.

An example — converting to ASCII

It is often necessary to convert a number in a register, ranging from 0 to 15, into an ASCII digit, from 0 to F. The problem is made clearer by examining the ASCII codes corresponding to each possible value:

Number	0	1	2	3	4	5	6	7	8
Symbol	0	1	2	3	4	5	6	7	8
ASCII	48	49	50	51	52	53	54	55	56

Number	9	10	11	12	13	14	15
Symbol	9	A	B	C	D	E	F
ASCII	57	65	66	67	68	69	70

The conversion looks easy up to 9 — simply add 48 on to the number. However, after 9 there is a jump in the ASCII equivalent of 7, then it gets back into step. What's required is thus:

If the number is >9 then add 7 to it
Add 48 (=ASCII for 0)

To code this in 68008, it has to be decided which register should hold the number. For convenience I chose to use D1 (choosing an address register would make life very much more difficult — try it if you like). The section of code to do the above turns out to be

```
            CMP.B     #9,D1
            BLS       DONTADD
            ADDQ.B    #7,D1
DONTADD     ADD.B     #"0",D1
```

Do you see the way the BLS works out whether or not to add the extra 7? The 'add 7' is skipped over if D1 is less than or the same as 9. This is not a complete piece of code on its own, so I've given no hex to enter into the QL. It is used in a short while though, to print out numbers in hex.

How they are calculated

Don't read this unless you want to, but I've included it for reference. Don't bother even thinking of learning this — it would be very difficult, and in any case there is absolutely no point. The table shows just what combinations of condition codes produce the extra conditions.

Condition	
HI	C=0 AND Z=0
LS	C=1 OR Z=1
GT	(N=1 AND V=1 AND Z=0) OR (N=0 AND V=0 AND Z=0)
GE	(N=1 AND V=1) OR (N=0 AND V=0)
LT	(N=1 AND V=0) OR (N=0 AND V=1)
LE	(Z=1) OR (N=1 AND V=0) OR (N=0 AND V=1)

Looping using DBcc

This is a very useful instruction for creating loops, using a data register as a counter. The 'cc' refers to a condition, and it has to be followed by a data register, then a label. The conditions allowed are the 14 described previously, together with two additional instructions — T for True, and F for False. The most common form of the instruction is

```
DBF D0,LOOP
```

What this means is 'decrement D0 and branch until False or until D0=−1'. As the condition false can never be met, this translates to 'decrement D0 and branch until D0=−1', so it is ideal for doing something a set number of times. As the loop finishes when the counter reaches −1, the initial value of the counter should be one less than the number of times round the loop. As an example, there follows a program that scrolls the screen up one line. To do this, the section of memory from $20080 to $27FFF has to be moved back in memory $80 bytes, and DBF is ideal for this.

Listing 4.1: Screen Scroll

```
303C7F7F          MOVE.W #$7F7F,D0          set count
207C0002          MOVE.L #$20080,A0         first location
0080
1150FF80 LOOP     MOVE.B (A0),-128(A0)      move a byte
5288              ADDQ.L #1,A0              increment A0
51C8FFF8          DBF    D0,LOOP            do whole screen
7000              MOVEQ  #0,D0              ready for BASIC
4E75              RTS                       and exit
```

A total of $7F80 bytes have to be moved, so this value less 1 is put into the count register, D0. (A word MOVE can be used, as DBcc only decrements the lowest 16 bits of the data register, and the test for −1 is also only done on the lower word.) The initial value of $20080 is put into register A0, and then the MOVE instruction does the job of moving the byte back in memory. It uses indirect addressing for the source, and indirect addressing with displacement for the destination. The detail of the next instruction, ADDQ, will be explained later, but for now just take it as read that it adds 1 to A0. The next instruction is the DBF, which does the hard work of decrementing D0, and going to LOOP until D0=−1, when it returns back to BASIC.

Although False is the usual condition in DBcc instructions, any of the others can be used. If the condition is met, then the loop will be exited prematurely, and the data register will not be decremented at that time. It makes no sense to use DBT, as the condition will always be met (as it is True), and will have no effect. (It is useful to know that some assemblers can accept DBRA as an alternative for DBF.)

Subroutines

There are two ways of calling subroutines on the 68008, both using a similar idea. They are JSR, for 'Jump to Subroutine', and BSR, for 'Branch to Subroutine'. When executed, the address following the instruction is put on the current stack, and the desired routine jumped to. For example,

```
        BSR PRINT   go to PRINT
REST    .........   more instructions
```

When the BSR gets executed, the value of REST is put on the stack, then control will pass to location PRINT. When the routine PRINT has finished, it returns to location REST with the instruction RTS — return from subroutine. This is the same RTS we use to get back to BASIC after a CALL, as the code you execute with a BASIC CALL instruction is just another subroutine to the system. What RTS does is to remove the top item from the stack, then pass control to it.

BSR, as with the other branch instructions, can only refer to locations within 32767 bytes. For longer subroutine calls, JSR has to be used, which can be followed by one of many addressing modes. The standard form is

JSR ROUTINE

where ROUTINE is expressed using absolute long addressing, but this form is not position independent, as BSR is. Many addressing modes can be used in a JSR instruction, though there are limitations. For full details, see the relevant page in Chapter 8.

The CLR instruction

This stands for 'clear', and is a fast and memory-efficient way of zeroing registers and memory. It can be byte, word or long in size, and the number of bits zeroed correspond to the size of the operation. Most of the addressing modes can be used to indicate what requires zeroing, and some are shown:

CLR.B D0	clear lowest 8 bits of register D0
CLR.W (A1)	clear lowest 16 bits of location A1
CLR.L FLAG	clear 32 bits at location FLAG

The forbidden addressing modes are address direct (so CLR.L A0 is not allowed), both PC modes, and immediate mode. As has been mentioned, register D0 must be 0 on return from BASIC, and the fastest way to do this is with the instruction CLR.L D0 which has an opcode of $4280.

More arithmetic

We have covered SUB and CMP, and their associated forms, and there is one more main arithmetic instruction left — that of ADD.

The ADD instruction

There are no prizes for guessing what this one does! It adds something to something else, the 'somethings' depending on the type of ADD. The types, along with their general forms are:

ADD (address),Dx	add contents of address to register
ADD Dx,(address)	add register to contents of address
ADDA (address),Ax	add contents of address to register
ADDI #(data),(address)	add data to contents of address
ADDQ #(data),(address)	fast form of ADDI

The range of additional types of ADD is similar to those for SUB, with similar rules, namely:

ADDA cannot be byte-sized, does not affect the condition codes, and always sign-extends its parameters to 32 bits; and ADDQ can only add the values 1 to 8 inclusive. The condition codes are affected in the usual way (except ADDA), though overflow is caused slightly differently — an overflow in ADD is caused by a sign-change when it shouldn't, eg ADD.B #8,D0 if D0=125, the result is 133, which is correct, but the sign has changed — it can be thought of as −123 as bit 7 is set.

Logical operations

There are three main logical operations — AND, OR, and Exclusive-OR, which share a similar set of instructions.

The AND operation

A logical AND takes the bits of two numbers, then ANDs them together. It is equivalent to the && operation in SuperBASIC. There are two general forms of the instruction:

AND (address),Dx and AND Dx,(address)

The first takes the contents of the address, logically ANDs it with the data register, and puts the result back into the data register. It can be any size, and only the relevant number of bits of each parameter are used.

The second form is similar, but the result of the operation is put back into the address. Here are some examples:

```
AND.B   #$0F,D1        D1=D1&&15
AND.W   D3,RESULT      POKE_W
                              RESULT,PEEK_W(RESULT)&&D3
```

There is another sort of AND, that of ANDI, for And Immediate. It has the form

ANDI #(data),(address)

and its main use is to logically AND memory with immediate data. The size of the data matches the size of the instruction. A good assembler will automatically decide which type of AND is necessary.

The final two types of AND are to alter the status register:

ANDI #(data),CCR and ANDI #(data),SR

The first has a data size of eight bits, and ANDs the data with the condition codes, placing the result back into the CCR. The second has a data size of 16 bits, and ANDs it with the contents of the complete status register, placing the result back in the SR. This is a privileged instruction, because of its potential power, and will only execute in supervisor mode.

OR and EOR instructions

The OR instruction does a similar operation to the SuperBASIC | | function. If either bit in the two numbers is set, then the resultant bit will be set, else it will be reset. The EOR instruction (for Exclusive OR) does a similar operation to | | in SuperBASIC. If both bits in a number are 1, then the resultant bit will be 0, or if one bit is 1, the result will be 1. There is a similar range of these functions to AND, namely

```
OR (address),Dx
OR Dx,(address)
ORI #(data),(address)
ORI #(data),CCR
ORI #(data),SR              (privileged instruction)

EOR Dx,(address)
EORI #(data),(address)
EORI #(data),CCR
EORI #(data),SR             (privileged instruction)
```

Note that there is one missing though — EOR (address),Dx is not permitted. You have to use an intermediate data register, eg

```
MOVE.B MASKS,D0
EOR.B D0,D2
```

instead of

```
EOR.B MASKS,D2
```

Shifts and rotates

The 68008 has four types of shifts and rotates, all in both directions, and they are:

```
LSR     logical shift right
LSL     logical shift left
```

ASR	arithmetic shift right
ASL	arithmetic shift left
ROR	rotate right
ROL	rotate left
ROXR	rotate with extend right
ROXL	rotate with extend left

What they do is move all the bits of an operand in a certain direction, though the exact details depend on the instruction type, best illustrated by a diagram (see **Figure 4.1**).

With all of these, any size can be specified, and only the expected number of bits are affected. The way in which the number of rotations can be specified, and what parameter is operated on, does vary, depending on the operation. These methods are the same for all the instructions, but in these examples ASL is used:

ASL Dx,Dy may be byte, word or long, and the number of times register Dy is shifted depends on the contents of register Dx, MOD 64).

ASL #(data),Dx may be byte, word or long, the number of shifts determined by immediate data, from 1 to 8 inclusive. Register Dx is the operand shifted.

ASL (address) this can be word only, and the contents of the address are only shifted once.

Arithmetic shifting is most used for signed numbers, and logical shifting for unsigned numbers. Rotate with extend is mainly used for high-precision maths operations.

The limit of eight on the number of operations when specified with immediate data is a nuisance, and there are two ways of getting around it. You can either put the count in another data register, or do an immediate count version a couple of times. For example, supposing you wanted to logically shift D3 word 12 times to the right, you could do either

```
MOVE.B #12,D0
LSR.W D0,D3          using D0 as the count
```

or

```
LSR.W #8,D3          maximum allowed
LSR.W #4,D3          then do again, giving a total of 12
```

The former method is preferable for large counts, while the latter is useful if register usage is limited, and there is no spare data register.

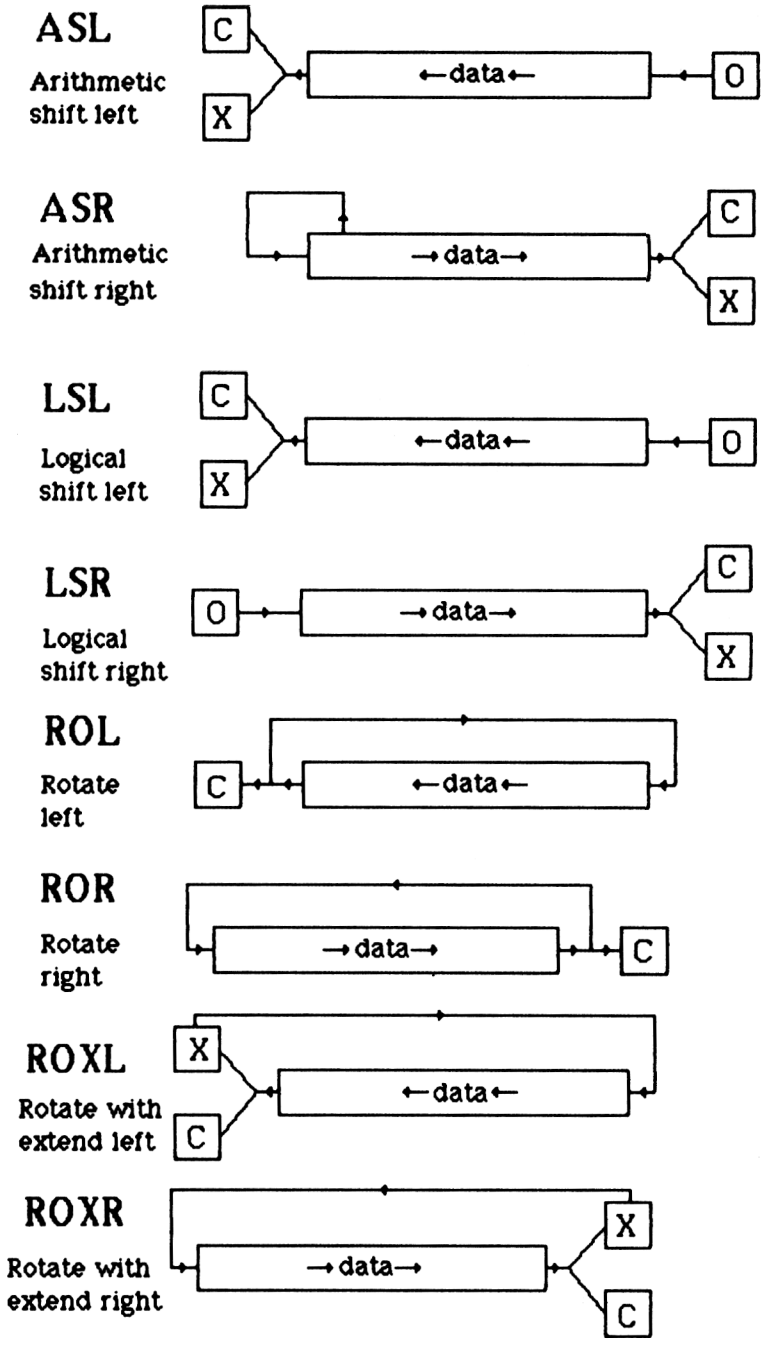


Figure 4.1: Shifts and Rotates.

Printing in hex

As a useful exercise, there follows a program that uses many of the instructions we've come across so far — a routine to print the value of a register in hex, though the actual mechanics for getting a character on the screen have not yet been covered.

Firstly, let's think about what is involved. We have to take each nibble (four bits), starting at the most significant, convert it to a single ASCII digit from 0 to F, print it, then do the same for all eight nibbles, printing the complete hex value. Converting a register value of 0 to 15 into an ASCII value of 0 to F has already been covered, so the main parts to be done are the 'extracting the nibble', and counting for eight digits. The complete subroutine looks like this:

Listing 4.2: Hex Printing

```
                    prints value of D1 as 8 digit hex

7007        HEX8    MOVEQ   #7,D0                  set count
E999        LOOP    ROL.L   #4,D1                  rotate nibble
2F01                MOVE.L  D1,-(A7)               save D1 on stack
0201000F            AND.B   #$0F,D1                D1=0 to 15
0C010009            CMP.B   #9,D1                  convert to ASCII
6F02                BLE     DIGIT
5E01                ADDQ.B  #7,D1
06010030 DIGIT      ADD.B   #"0",D1
6108                BSR     PRINT                  print the character
221F                MOVE.L  (A7)+,D1               restore D1
51C8FFE6            DBF     D0,LOOP                do all digits
4E75                RTS                            and finish

                    prints chr$(d1) onto screen
                    without corrupting any registers
                    (Don't worry how it does it yet)
48E7D0C0 PRINT      MOVEM.L D0-1/D3/A0-1,-(A7)
207C0001            MOVEA.L #$00010001,A0
0001
76FF                MOVEQ   #-1,D3
7005                MOVEQ   #5,D0
4E43                TRAP    #3
4CDF030B            MOVEM.L (A7)+,D0-1/D3/A0-1
4E75                RTS                            end of subroutine
```

It starts off by assuming the register value to be printed lies in D1. Register D0 is going to be the counter, so it gets initialised to the number of digits −1, which is 7. (The minus one is necessary because we're going to use DBF.) We then enter the loop, in which D1 (long) is rotated left four times. To start with, this puts the highest nibble (at bits 28–31) into the lowest four bits, and rotates everything else in the register up one nibble. The MOVE saves the value of D1 on the stack for later, then the AND makes D1 hold 0 to $0F inclusive, which then gets converted to its ASCII equivalent. (This corrupts the value of D1, which is why we saved it previously.)

After the conversion, the BSR PRINT calls the subroutine that sends character D1 to the screen, then D1 gets restored to its old value, and the DBF makes sure the loop goes round eight times. Each time, the next nibble gets rotated around into the lowest four bits, converted and printed, until the whole number has been printed. Don't concern yourself with PRINT for the moment — it is fully covered in Chapter 7, but take it as read that it sends a character to the screen, without corrupting any registers.

The subroutine itself cannot be used directly from BASIC as it stands — it will work perfectly, but generates an error message when it finishes, because the final value of D0 will be −1. It is intended to be called by other bits of program, so D0 is not explicitly cleared.

It can be extended to print values in hex of 8 or 16 bit quantities too. For 8 bit data, with two hex digits, the initial value of D0 has to be set to 1, and D1 rotated initially so what was originally in the lowest four bits goes into the highest. For 16 bit data, D0 has to be 3 to start with, and its low and high words swapped around for the data to be in the right place. This can be done quickly with the SWAP instruction. The necessary extra instructions are:

Listing 4.3: Printing 8 and 16 Bit Values

(must immediately precede Listing 4.2)

```
                    8-bit data - 2 hex digits

7001        HEX2    MOVEQ   #1,D0                  set count
E099                ROR.L   #8,D1                  rotate value
6008                BRA     LOOP                   and print it

                    16-bit data - 4 hex digits

7003        HEX4    MOVEQ   #3,D0                  set count
4841                SWAP    D1                     move data
6002                BRA     LOOP                   and print it
7007        HEX8    MOVEQ   #7,D0                  rest of routine
                    .........
                    .........
```

(The hex bytes assume that the previous routine directly follows these instructions.) This now gives three subroutines for the price of one. They are HEX2, for 8 bit values, HEX4, for 16 bits, and HEX8, for 32 bits. Having done this you could always try for something a little more adventurous — how about printing them in decimal? (This is very tricky indeed, and the faint-hearted shouldn't try it — anyway there's a nice QDOS routine that does it all for you in the ROM!)

CHAPTER 5

Further Instructions and Passing Parameters

One of the instructions carried over from previous Motorola processors is LEA, which stands for 'Load Effective Address'. At first sight it seems identical to the MOVEA instruction, but there is a subtle difference, best illustrated by this example:

```
MOVE.L LABEL(PC),A1          LEA LABEL(PC),A1
```

What the MOVE does is to read the *contents* of location LABEL, and put them in register A1. The similar LEA instruction places the *value* of LABEL into A1, not the contents of the location. The difference between these two is a well-known source of confusion to newcomers to the 68000 series, which is understandable. LEA is like MOVE, but the value of the address is placed in the selected address register. To get the same effect as the above LEA instruction, you could do

```
MOVE.L #LABEL,A1
```

but there is an important difference. The LEA form, as it uses PC mode, is position independent, whereas the MOVEA form, using direct addressing, is not.

Probably the main use of LEA is in getting around the restriction that prevents PC mode being used as the destination in MOVE instructions. For example, suppose you wanted to do

```
MOVE.L D3,STORAGE(PC)
```

which is not allowed, you could get around the restriction with

```
LEA STORAGE(PC),A1
MOVE.L D3,(A1)
```

The LEA sets A1 to the value of STORAGE (using PC mode), and then the MOVE does the transferring of data. LEA can also be used to add

things together faster than can be done with the ADD instruction. For example, suppose you wanted to calculate the sum of A1,A2 and 9 and put the result in A3, it could be done with

```
MOVE.L A1,A3        A3=A1
ADDA.L A2,A3        A3=A1+A2
ADDA.L#9,A3         A3=A1+A2+9
```

but a much faster and neater way is by using LEA, with LEA 9(A1,A2),A3 (compare this with MOVE.L 9(A1,A2),A3, which takes the contents of memory location (A1+A2+9) and puts it in A3).

LEA does differ markedly from MOVE in some respects, though. Firstly, its size is always long, and secondly, the destination is always an address register. There are also limits on which addressing modes can be used with LEA. Chapter 8 lists them all, but the most notable illegal mode is immediate data — LEA #100,A2 simply doesn't make sense. If you want to put a value of 100 into A2, you can use either LEA 100,A2 or MOVE.L #100,A2.

The NOP instruction

This is probably the simplest 68008 instruction to understand of all. It stands for 'no operation', and does just that — nothing. It is normally used in things like delay loops, to use up time, and to obliterate instructions when debugging programs.

The EXG instruction

This is another easy one. It stands for 'exchange registers', and does exactly what you would expect — it swaps the values of any two registers (and is always long in size). For example,

```
EXG D3,A0
```

would swap the values of D3 and A0 around. Despite its simplicity, it can get confused with two other completely different instructions — EXT, and SWAP, so beware.

The NOT instruction

This one 'inverts' the contents of the specified address. For example, if location $30000 contained $1F initially, after the instruction

```
NOT.B $30000
```

it would contain the inverse of $1F, which is $E0. The inverse of a number means that the state of all the bits are swapped, and this can be seen by the examination of the binary forms of these two numbers:

```
$1F     0001    1111
$E0     1110    0000
```

As well as inverting memory, it can also be used on data registers (but not address registers), and can be any size eg

```
NOT.W D2
```

This instruction should not be confused with the next one, though.

The NEG instruction

This is similar to NOT, but instead of the memory or register contents being inverted, they are two's-complemented. Thus $1F would get changed to −$1F, which is $E1 (from 256–31). It can be any size, so a NEG operation on a word value of 123 decimal would give a value of 65536–123=65413= $FF85. As with NOT, data registers can be two's-complemented, but not address registers.

The SWAP instruction

Not to be confused with EXG, this swaps the high and low words of a data register around. It is not possible to swap anything but data registers though.

Passing parameters using CALL

A popular use of machine code on any computer is not just for stand-alone programs, but for adding additional features and functions to the machine's BASIC, and the QL is no exception. You will usually find that you need to pass the values of one or more BASIC variables on to the routine, and will possibly want values returned by the routine. The former at least is made easy by the way SuperBASIC does its CALL command.

As was covered in Chapter 3, CALL should be followed by the location at which your machine code program starts. However, you can follow the location with up to 13 numeric parameters, separated by commas, eg

```
CALL 261120,10,34,a,r(5)
```

If SuperBASIC finds such parameters following a CALL, it puts the values into the 68008 registers just prior to calling your machine code. The order it puts the values into the registers is:

D1, D2, D3, D4, D5, D6, D7, A0, A1, A2, A3, A4, A5

The missing registers are D0, which is used as an error number, A6, which holds an important address for SuperBASIC, and A7, which is the user stack pointer. Each parameter can range from −2147483649 to 2147483648 (or more clearly 0–$FFFFFFFF as signed integers). String parameters are not allowed, and attempting to use them will crash early versions of the QL. If you specify more than 13 parameters, the extra ones will be ignored.

If you want to pass just one parameter to a routine, don't choose something silly like A1, which requires a line like

CALL location,0,0,0,0,0,0,0,0,value

which is very easy to mistype. Make life easy and use D1, translating the above into a more manageable

CALL location,value

even if you have to add to the front of your program the instruction

MOVEA.L D1,A1

While we are discussing CALL, there are a few other points to be made about it. Firstly, if you're going to use A6, save its initial value and restore it before you RTS to BASIC, else you will crash it out. Secondly, the same rule applies to A7 — preserve its value if you're going to use a different location for the user stack. Also, if you go into supervisor mode, remember to get back into user mode before returning to BASIC.

The CALL command does allow values to be passed to it, but unfortunately doesn't allow values to return from it. The way to do this is to reserve a suitable number of bytes at the end of your program, and MOVE the data you want to pass back to BASIC into it before you RTS. Then the BASIC program can PEEK the appropriate locations to see what the data was. As an example of this, there follows a program that passes a short string back to BASIC, revealing which version of QDOS is installed in the QL. It does this by using a 'trap' instruction (traps are more fully covered in Chapters 6 and 7) to call a QDOS routine.

Listing 5.1: Which QDOS Version

```
4280                CLR.L   D0                  get ready for the trap
4E41                TRAP    #1                  call QDOS
41FA0006            LEA     PARAMS(PC),A0       A0=PARAMS
2082                MOVE.L  D2,(A0)             store D2
4E75                RTS                         and back to BASIC
00000000 PARAMS     DS      4                   storage location here
```

Firstly D0 is zeroed, which is necessary for this particular trap, then the TRAP #1 instruction done. What this does is basically a special system call to the ROM, asking it for details about the state of the machine. Various information is returned, but the type number comes back in register D1 long, in the form of four ASCII bytes. The LEA and the MOVE together store this value at location PARAM, and the RTS returning to BASIC is executed. (It is not necessary to zero D0, as the trap does it for you.) To use this, a program like this is needed:

```
1000  start=?????:REMark wherever the m/c starts
1010  CALL start
1020  PRINT "The version number of this QL is";
1030  FOR i=0 to 3
1040  PRINT CHR$(start+12+i);
1050  NEXT i
1060  PRINT
```

Depending on your QL, this returns a value such as 1.02 or similar.

CHAPTER 6

Exception Processing, Traps and Interrupts

A major feature of all the 68000 series processors is their *exception processing* abilities. An exception is an event that causes a special sequence to occur, usually involving a subroutine. Unfortunately the QL firmware does not allow the full range of exceptions, which is a pity. For this reason, I shall describe the standard features, and also the limitations placed on each by the QL.

The vector table

Crucial to exception processing is a 1024 byte table of vectors, running from location $00000 to $003FF inclusive. The table consists of 256 vectors, each taking a long word, ie four bytes. Each vector has a number, starting from 0, so that the vector at $00000 is number 0, the one at $00004 is number 1, and so on. The full table is defined as laid out in **Table 6.1.**

It is normal practice with 68000 series machines to have RAM in this vector area, for maximum flexibility. However, at switch-on, RAM contents are undefined, so some simple electronics are required to switch a ROM into this area just long enough to get the system started properly. To cut costs on the QL, though, this was ruled out, and it was decided to place ROM permanently in this area. Regrettably, it was also decided to use some of the vector table not only for vectors, but for holding actual instructions, thus 'destroying' the vectors at several points. It is this fact that makes many of the vectors above 'unusable'.

General exception processing

Although there are many different types of exception, most follow certain rules when they occur — these are:

an internal copy of the SR is made, for later use
the S bit is set, entering supervisor state
the T bit is reset, disabling trace (see later)
the interrupt bits may be altered, depending on the exception

Table 6.1

Location (hex)	Decimal	Number	68000 series use	QL Usage
000	0	0	RESET vale of SSP	same
004	4	1	RESET value of PC	same
008	8	2	Bus error	ignored
00C	12	3	Address error	user vector
010	16	4	Illegal instruction	user vector
014	20	5	Divide by zero	user vector
018	24	6	CHK instruction	user vector
01C	28	7	TRAPV instruction	user vector
020	32	8	Privilege error	user vector
024	36	9	Trace	user vector
028	40	10	Line 1010 emulator	unusable
02C	44	11	Line 1111 emulator	unusable
030	48	12	Undefined	unusable
034	52	13	Undefined	unusable
038	56	14	68010 Format error	unusable
03C	60	15	Uninitialised Interrupt	unusable
040	64	16	all	all
to 05F	to 95	to 23	Undefined	unusable
060	96	24	Spurious interrupt	ignored
064	100	25	Interrupt level 1	ignored
068	104	26	Interrupt level 2	system interrupt
06C	108	27	Interrupt level 3	ignored
070	112	28	Interrupt level 4	ignored
074	116	29	Interrupt level 5	ignored
078	120	30	Interrupt level 6	ignored
07C	124	31	Interrupt level 7	user vector
080	128	32	Trap #0	QDOS call
084	129	33	Trap #1	QDOS call
088	130	34	Trap #3	QDOS call
08C	131	35	Trap #4	QDOS call
090	132	36	Trap #5 to	all user
to 0BF	to 191	to 47	Trap #15	vectors
0C0	192	48	all	all
to 0FF	to 255	to 63	Undefined	unusable
100	256	64	all	all
to 3FF	to 1023	to 255	User Interrupt Vectors	unusable

the vector number is calculated according to exception
the old value of the PC is put onto the SSP (Long)
the old value of the SR is put onto the SSP (Word)
the value of the vector is read from the table, and control passes to it

This may seem complicated, but all the programmer has to know is that the machine goes into supervisor mode, PC is stacked, SR is stacked, and a jump according to the vector made. Whenever an exception handler has finished, it returns to whatever triggered it by doing the RTE instruction, which stands for 'return from exception'. What it does is remove the old value of SR from the stack, then the old PC, and jumps to it.

Having covered the table, we'll now look at each exception in turn.

0 & 1 — RESET

After a switch-on, the 68008 is in supervisor mode, and it looks at vectors 0 and 1 for its initial information. Vector 0 holds the initial value of the supervisor stack pointer, and vector 1 holds the start value of the program counter. On the QL the initial SSP is $40000 (the very top of RAM+1) while the initial PC value varies depending on the ROM, but normally lies between $014E and $0168, ie the bottom of the system ROM.

A reset can also occur when desired by external hardware, but this should never happen on the QL. Note that this is the only vector not to save PC and SR on the stack.

2 — Bus Error

This is an error determined by external hardware, and should never occur on the QL. Its normal use is in conjunction with a device known as a memory management unit (MMU), which can stop programs accessing certain parts of memory that they are not entitled to. With a bus error, four additional words are pushed on the stack in addition to the PC and SR. The additional information (in the order in which it is stacked) is: the instruction register; low word of the address accessed; high word of the address; and a special format word.

3 — Address Error

This is an error that occurs if a word or long word access is attempted at an odd-numbered address. Such an exception on the QL may or may not cause a system crash, but it can be re-defined by the programmer — the method will be described later. An address error also stacks similar additional information to a bus error.

4 — Illegal Instruction

There are many undefined opcodes in the 68008 instruction set, as well as an instruction ILLEGAL, which has an opcode of $4AFC. Other

instructions defined as illegal are byte-sized address register direct instructions, and attempts to use illegal addressing modes. In particular, trying to use either PC modes in the destination of a MOVE will force this exception. This exception will generally cause a crash on the QL, but may be user-defined.

5 — Divide by Zero
When using the divide instructions DIVU and DIVS, if the source data is zero this exception will occur. It is normally ignored on the QL, but may be user-defined.

6 — CHK Instruction
This exception is called if the register in a CHK instruction is out of bounds. It is normally ignored on the QL, but may be user-defined.

7 — TRAPV instruction
This exception occurs if the TRAPV instruction is executed and the overflow (V) flag is set. It is normally ignored on the QL, but may be user-defined.

8 — Privilege Violation
If the 68008 is in user mode, and a privileged instruction executed, this exception will occur. It is usually ignored on the QL, but may be user-defined. The privileged instructions are those that alter the status register, RTE, RESET, and STOP.

9 — Trace Exception
Bit 15 of the SR is the trace bit and, if it is set, then trace exception processing will occur. This means that after every instruction is executed, the trace exception handler is executed. As you may remember, during any exception the trace bit is reset, so you don't get the exception handlers themselves being traced. You can do many things with trace, such as single-stepping. This lets you execute instructions one at a time, by pressing a key, say. Trace mode can be enabled with the instruction

```
ORI #$8000,SR
```

and disabled with

```
ANDI #$7FFF,SR
```

Both these instructions are privileged. The exception is normally ignored on the QL, but may be user-defined.

10 — Line 1010 Emulator
All opcodes with bits 12 to 15 as binary 1010 are unused, and if executed force this exception. It is thus possible to add your own instructions to the set, but regrettably this vector is unusable on the QL, as it is overriden by instruction codes.

11 — Line 1111 Emulator
This is similar to the above exception, but with bits 12 to 15 equal to 1111. It too is unusable on the QL, for a similar reason. On the 68020 processor, these instructions are interpreted by the 68881 floating point maths co-processor.

12 & 13 — Undefined
These two vectors are unassigned vectors, and reserved for future enhancements by Motorola. Whatever the enhancements are, the standard QL won't support them, as these vectors are unusable.

14 — 68010 Format Error
This is an exception triggerable only by the 68010 processor, caused by an invalid format code after an exception. It is irrelevant to the QL, and the vector is unusable anyway.

15 — Uninitialised Interrupt
This is a hardware-induced exception, which should not occur on the QL, which is just as well, as this vector is unusable.

16 to 23 — Undefined
These are more vectors reserved for enhancements, and again unusable on the QL.

24 — Spurious Interrupt
This is a hardware-induced exception, caused when an external device fails to do as it should during an interrupt. On the QL, it is ignored.

25 to 31 — Interrupt Vectors
The 68000 series processors all support seven levels of interrupts, except the 68008 — of the seven possible levels, the 68008, and thus the QL, only supports levels 2, 5 and 7. The current level of interrupt is determined by bits 8 through 10 of the SR, and can vary from 0 to 7. On the QL, these values correspond to the following allowed levels:

SR BITS 8–10	ALLOWED LEVELS
0	2,5,7
1	2,5,7
2	2,5,7
3	5,7
4	5,7
5	5,7
6	7
7	7

It follows from this that the interrupt with the lowest priority is level 2, next is level 5, and the highest priority goes to level 7. You may also note that level 7 interrupts can never be disabled, and are thus equivalent to 'non-maskable interrupts' on other processors.

The unavailable interrupt vectors, ie nos 1, 3, 4 and 6 would all do nothing if executed on the QL. Level 2 is the main system interrupt, and does an awful lot of things, including scanning the keyboard, and multi-tasking. Level 5 interrupts are ignored, while level 7 interrupts are normally ignored, but may be user-defined. While an interrupt exception is executing, the interrupt mask in the SR is altered so that lower priority interrupts are inhibited.

Level 2 interrupts are triggered by the electronics within the QL, the timing and use of which are determined by one of the custom ULAs. Level 7 interrupts are left to be triggered by additional hardware.

32 to 47 — Trap Vectors

There are sixteen TRAP instructions in the 68008 set, denoted by being followed by a hash sign, then their number. They vary from TRAP #0 through to TRAP #15, and are program-induced exceptions. Their approximate equivalents in other processors are 'software interrupts' on the 6809, and 'Restarts' on the Z80 and 8080. They are really special subroutines, though with a few differences. On the QL the first five are defined to be QDOS system calls, so that the ROM may be changed without losing compatibility with previous versions (so long as the traps functions aren't changed of course). Traps 5 through 15 are normally ignored, but may be user-defined.

So what do traps 0 through 4 do? Well, a complete description is beyond the scope of this book, but briefly they do the following:

Trap #0 is the simplest, and switches you into supervisor mode. There are additional rules to beware of when running programs in supervisor mode, mainly to do with the stack. In particular, A7 points to a different area to that of the user A7, and there is not very much spare space below it. This means that there is a practical limit of around 64 bytes that can safely be

used on the supervisor stack. The easiest way to get back to user mode is with the instruction

```
ANDI #$DFFF,SR
```

Trap #1 is the 'QDOS manager trap'. It is used to control various important machine resources, and the exact effect depends on the value of register D0. Additional parameters may be passed and returned in D1–D3 and A0–A3.

Trap #2 is the QDOS basic I/O trap, again using D0 to determine the operation.

Trap #3 is the QDOS trap for more complex I/O, such as screen, colour, window and file-handling.

Trap #4 is a trap for the BASIC command interpreter that converts passed parameters from their relative form to an absolute form.

The above information is not enough to use any of the traps except 0. If you wish to delve deeper into the delights of QDOS then I suggest you get hold of either Sinclair's own technical information, or a suitable independent publication (such as my next book from Sunshine, *The QDOS Companion*). However, some QDOS traps are described later.

48 to 63 — Undefined
These are more vectors that are officially reserved for future use, and unusable on the QL. However, there are many two-byte vectors at this point on the QL, starting from $000C0, running on past $00110, depending on the ROM version. These are a way of handling certain ROM routines while retaining compatibility if a ROM changes. To use them, indirect addressing is used in a JSR instruction, so for example if vector $C0 is required, it can be called with

```
MOVE.W $00C0,A0
JSR (A0)
```

What this does is to read the word contents of $C0 into A0, then call the routine. The example assumes A0 does not need to pass a parameter to the subroutine — if this is desirable then simply use a different address register. (This is practically the only use of absolute short addressing on the QL.)

64 to 255 — User Interrupt Vectors
These are intended for storing vectors on multiple-interrupt systems, but the QL handles its interrupts using a linked list, and thus has no need for these vectors. The first few are used as word-sized vectors for the system, though.

User-definable exception vectors

As has been mentioned previously, certain QL vectors may be re-defined, via RAM. The vectors, and their RAM numbers, are:

Ram	Number	Use
00	0	Address error
04	4	Illegal instruction
08	8	Division by zero
0C	12	CHK instruction
10	16	TRAPV instruction
14	20	Privilege violation
18	24	Trace vector
1C	28	Interrupt level 7
20	32	Trap#5
24	36	Trap#6
28	40	Trap#7
2C	44	Trap#8
30	48	Trap#9
34	52	Trap#10
38	56	Trap#11
3C	60	Trap#12
40	64	Trap#13
44	68	Trap#14
48	72	Trap#15

These RAM vectors are enabled by telling the system that you want it to look at your own table, in RAM. The table requires 76 bytes of RAM, and should be set up before you tell the system about it, to prevent nasty happenings. It's also a good idea to make all unused RAM vectors point to an RTE instruction, just in case they are triggered. When you've set up your table, you tell the system to use it by doing a QDOS system call, via TRAP#1. It requires the following parameters:

D0 byte — 7 — signals 'trap redirection'
D1 long — Job ID — normally −1
A1 long — start of RAM vector table

As with many QDOS calls, D1 must contain the relevant Job ID. This is because QDOS is multi-tasking, and different jobs may have different RAM vector tables. Normally, only one job is running, namely BASIC. The value of −1 is used to indicate 'the current job'.

An example of RAM vectors

When testing machine code programs, it can be very difficult to debug them if they simply cause the QL to 'hang' — in other words, the QL just sits there, in some form of infinite loop. I have found (the hard way!) that two of the most popular ways of generating apparent infinite loops are:

(i) Trying a word or long access on an odd address
(ii) Mis-coding an instruction, and using illegal addressing modes

By writing your own exception handlers for 'address error' and 'illegal instruction', you can improve your chances of finding errors such as these. There follows a complete program that sets up a RAM exception table so that either one of these programming errors will produce a suitable message on the screen, though will not necessarily leave the machine in a fully-functional state.

Both handlers use a QDOS vector to print the messages 'bad parameter' for an address error and 'not implemented' for an illegal instruction. Both these messages are system error messages, with numbers −15 and −19 respectively. The vector used is the one at $00CA, which requires only one parameter — namely the error code in register D0. The error message will be printed on channel 0, the lower screen. (Of course, if your wild program has inadvertently corrupted certain system variables then you won't see anything printed.) They each also try to recover from the error, by returning to the instruction after the one that caused the exception — this will work with some errors, but is not foolproof.

Listing 6.1: Trapping Program Bugs

```
43FA0048 SETUP    LEA     TABLE(PC),A1    A1=table start
41FA0026          LEA     ADDERR(PC),A0
22C8              MOVE.L  A0,(A1)+        store ADDERR in TABLE
41FA0032          LEA     ILLERR(PC),A0
22C8              MOVE.L  A0,(A1)+        store ILLERR in TABLE+4
41FA002A          LEA     OTHER(PC),A0
203C0000          MOVE.L  #16,D0          number of other vectors-1
0010
22C8      FILTAB  MOVE.L  A0,(A1)+        store OTHER in rest of table
51C8FFFC          DBF     D0,FILTAB       do all 17
43FA0028          LEA     TABLE(PC),A1
72FF              MOVEQ   #-1,D1          do job ID
7007              MOVEQ   #7,D0           signal 'RAM traps'
4E41              TRAP    #1              tell QDOS about it
4E75              RTS                     then back to BASIC
```

```
                    Address error exception handler

203CFFFF ADDERR     MOVE.L #-15,D0          'bad parameter'
FFF1
508F                ADDQ.L #8,A7            skip over extra words on stack
30790000 PRTERR     MOVEA.W $00CA,A0        get the vector
00CA
4E90                JSR    (A0)             call the 'print error message' routine
4E73        OTHER   RTE                     and end the exception

                    Illegal exceptiom handler

203CFFFF ILLERR     MOVE.L #-19,D0          'not implemented'
FFED
54AF0002            ADDQ.L #2,2(A7)         restart at next instruction
60EA                BRA    PRTERR           then print the message
          TABLE     DS     76               room for the RAM vector table
```

It works like this. Firstly, a RAM vector table must be set up, and this is at location TABLE. The first vector in it is the address error vector, so ADDERR gets MOVEd into it. Note the way auto-increment addressing has been used, to make it easier to set the table up. Into the next vector location goes ILLERR, then a loop makes sure all the other vectors are OTHER, which is a RTE statement. After the loop, the trap is done to tell the system to use the table. The address error exception handler is straightforward — D0 is set to an appropriate number for 'bad parameter', then 8 is added to the value of A7 — this is done to skip over the extra four words put on the stack. Next the vector is read from the table in the ROM, the 'print error message' routine called, and finally an RTE done. The illegal exception handler firstly sets D0 to −19 for 'not implemented', then 2 is added to location 2(A7). What this does is to increase the return address by two, so that the RTE will not go back to the instruction that caused the exception in the first place, but to the next one. Then a branch is made to PRTERR, to print the message and exit from the exception. The mnemonic DS is not a 68008 instruction, but an assembler instruction standing for 'define space'. It simply makes sure that no following instructions would get assembled into the area. (For Zilog fans, it is equivalent to the Z80 pseudo-op DEFS.)

As a simple test, try running programs with the instruction ILLEGAL in it, and try doing MOVE.L 21,(A0) which tests both exception handlers.

It is not 100% perfect, and as with all programs there is scope for possible improvements. How about getting it to print the instruction at which the exception occurred, or making it go into a deliberate infinite loop afterwards so that you can be sure of reading the message before your program continues and corrupts the screen?

CHAPTER 7
Using the Hardware and Firmware

The QL has many interesting hardware features, and in this chapter the most useful are described, including the screen and 8049 second processor. There are two main areas of the QL memory map that control this hardware — the screen memory, which is variable, and the I/O area starting at $18000.

The screen

The QL screen has two modes — 4 colour, with 512 × 256 pixels, and 8 colour, with 256 × 256 pixels and flashing. In both modes, 32K RAM is taken for the display, normally starting at $20000. The mode can be controlled by two methods — the proper way, using a QDOS trap, or the not so proper way, by directly accessing the I/O location of the screen register.

Regardless of mode, the screen memory organisation is basically similar, with the screen divided into 256 lines, each of 128 bytes. Each line of bytes is organised into 64 words, though the meaning of each word's contents depends on the mode. The top left of the screen corresponds to location $20000, and the memory is organised in the expected order — from left to right along each line, going from the top to the bottom, so for example the lefthand side of the second line is location $20080. The normal screen memory map is shown in **Figure 7.1**.

Although the actual screen size is 512 × 256 in mode 4, most televisions and some monitors are incapable of showing the whole screen. Normally some part of each extreme 'falls off the edge' of the viewable screen, so software has to compensate for it. This is the reason why the QL asks you to press F1 or F2 when you switch it on, so that the default windows can be set up to suit the display device. **Figure 7.2** shows how the default windows for both modes compare with the actual screen size, to give you an idea of what parts of the screen are not normally displayed. Note that all coordinates are based on the pixel coordinate system, which defines the screen to be 512 × 256 pixels, regardless of mode.

There are two ways of writing things on to the screen — either by one of the many QDOS traps, or by writing directly into screen memory. The advantages of the system traps are that they are relatively easy to use and

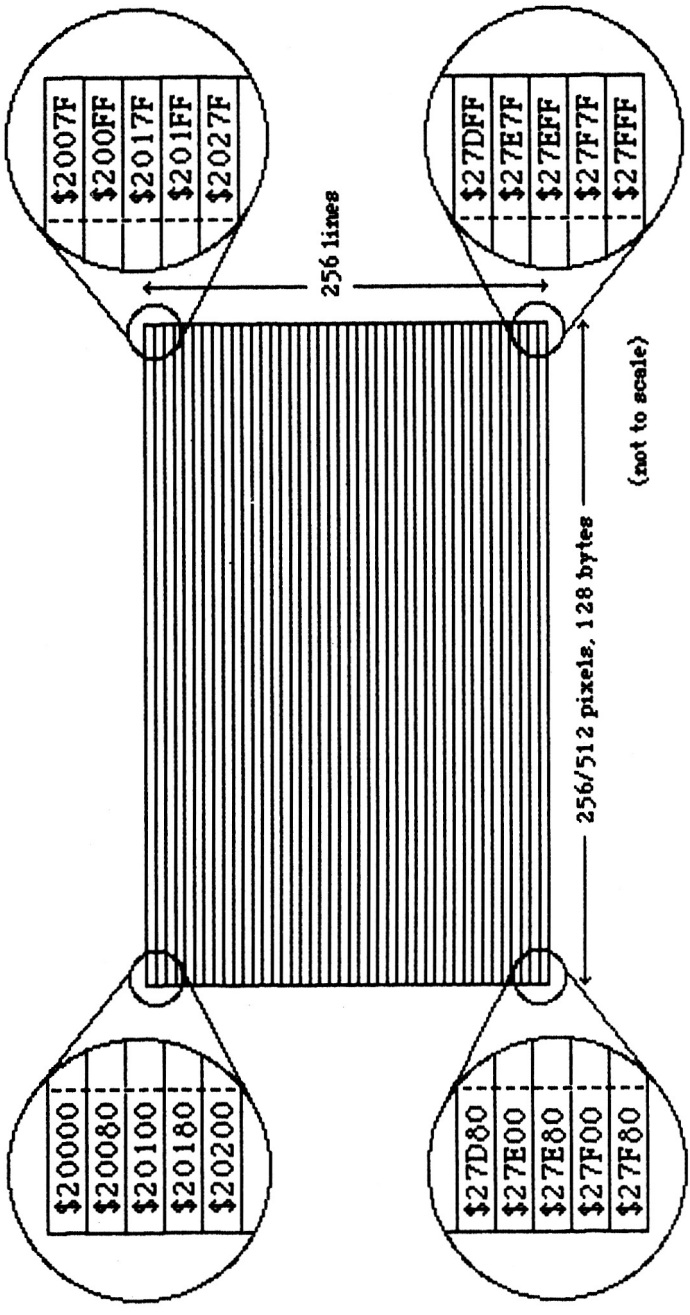


Figure 7.1: Usual Screen Memory Map.

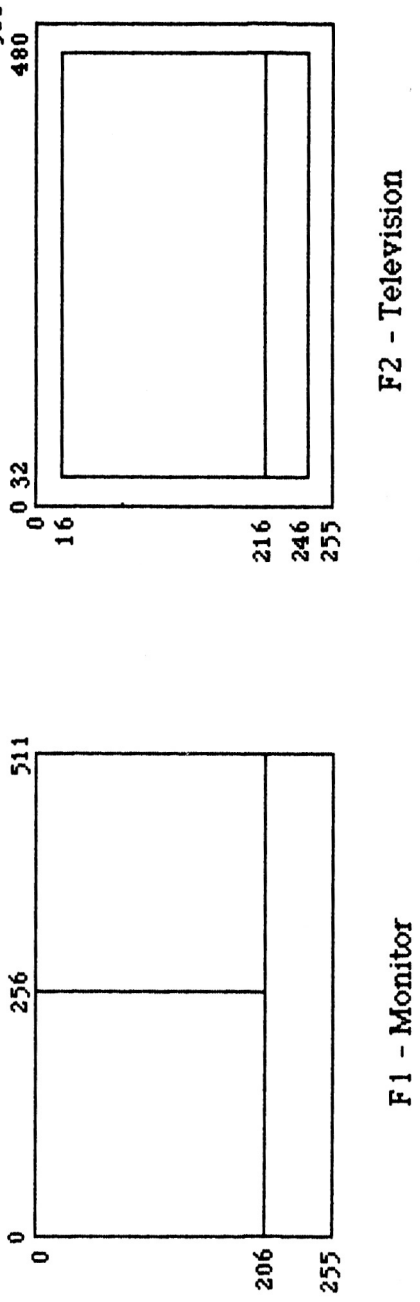


Figure 7.2: Default Window Sizes.

(almost) bug-free. The disadvantages are that they are comparatively slow to execute, and require QDOS to be usable. The latter reason may seem a bit strange, but later on we'll discover why — it's to do with a second possible memory location for the screen.

The contents of each word controls the colour of each pixel, and the flashing in 8 colour mode. The colour values are shown in this table:

Number	Binary	8 colour	4 colour
0	000	black	black
1	001	blue	black
2	010	red	red
3	011	magenta	red
4	100	green	green
5	101	cyan	green
6	110	yellow	white
7	111	white	white

It can be seen that the binary for each colour controls the mix of the primary colours — bit 0 determines blue (in 8 colour mode only), bit 1 determines red, and bit 2 green. In 4 colour mode, red and green together give white.

This use of each bit of the words depends on the screen mode as follows.

4 colour mode — high resolution

This is the simplest, with one word controlling eight pixels of the display. Each bit of the high byte controls the green content of each pixel, while each bit in the low byte controls the red content. If the same bit is set in both bytes of the word, then the resultant colour is white. If both are reset there is no colour — ie black. This is shown in **Figure 7.3**.

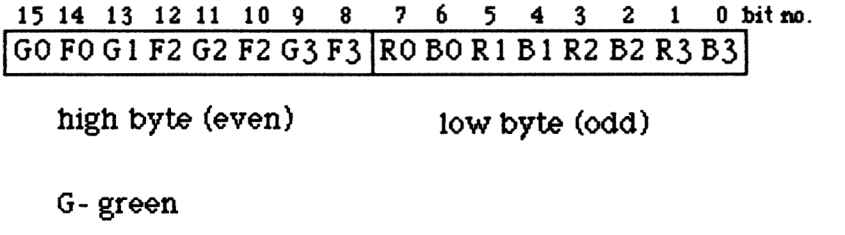


Figure 7.3: Word Arrangement in Four-colour Mode.

So, for example, if a screen word contains $53C6 then the corresponding pixel colours are red/white/black/green/black/red/white/green, from left to right. This is worked out by writing the binary form, as in **Figure 7.4**.

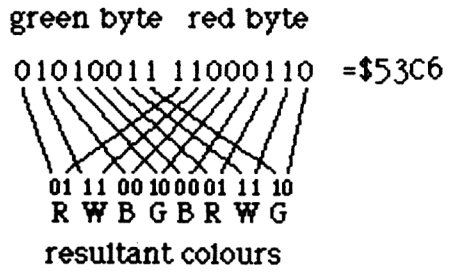


Figure 7.4.

8 colour mode — low resolution

In this mode one word controls the colour of four pixels, with the high byte determining green and flash, and the low byte determining red and blue content of each pixel. The flash facility works rather strangely, in a serial fashion. At the beginning of each pixel line, flash is off, and when a suitable bit is set, flash will go on, and stay on across the line until another flash bit is set, which will switch it off, and so on across the line. In this way, a set flash bit toggles the current flashing status, which does make it harder to use. The bit usage in each word is shown in **Figure 7.5**.

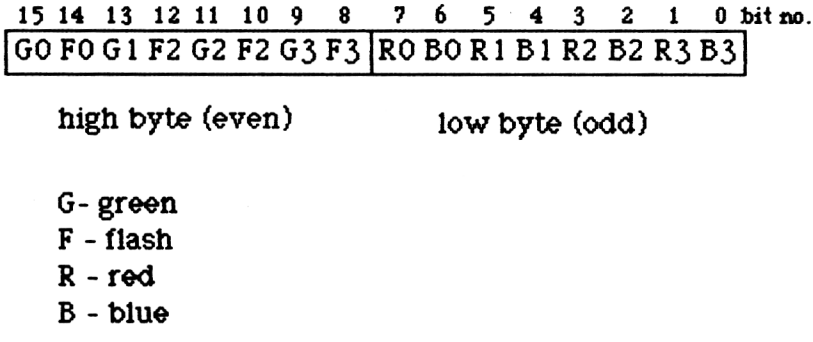

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 bit no.
G0 F0 G1 F2 G2 F2 G3 F3 R0 B0 R1 B1 R2 B2 R3 B3
high byte (even)
low byte (odd)
G- green
F - flash
R - red
B - blue

Figure 7.5: Word Arrangement in Eight-colour Mode.

So, if the word $288B will give pixel colours of red/green/yellow/magenta, from the binary (see **Figure 7.6** — for simplicity, all the flash bits are zero). The primary way of altering the screen mode is via a QDOS call with TRAP #1, with the following parameters:

D0 — $10 to select function
D1 — 0 for 4 colour, 8 for 8 colour
D2 — usually −1

The trap does several things — firstly it changes the mode by sending data to the hardware, then zeroes every byte in the display memory (ie turns it

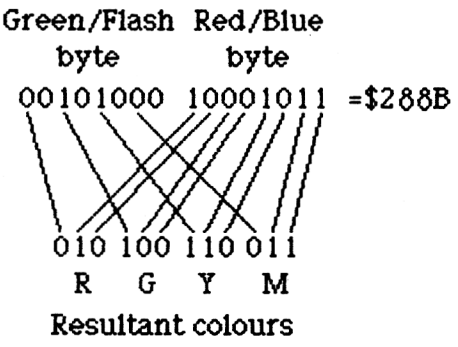


Figure 7.6.

black). Next it goes through all the screen channels and alters their parameters to suit the new mode, does a CLS for each window, and finally returns with D2.byte containing either 0 if F1 was pressed on switch-on, or 1 if F2 was pressed.

There is an alternative way to change screen mode, which is suitable if you aren't going to be using any QDOS screen calls, or if you want to do without QDOS altogether. The 'master chip status register' (or MCSR) is the long name for the memory location that controls the screen mode, and is at location $18063. The byte that gets written into it determines the mode, depending on certain bits, as follows:

BIT	FUNCTION
1	0 — turn display on, 1 — turns off
3	0 — 4 colour mode, 1 — 8 colour mode
7	0 — screen #0, 1 — screen #1 (see later)

Don't worry about bit 7 at the moment — normally leave it zeroed.

Plotting points

Although there is a QDOS trap to plot points, it is a very useful exercise to write your own routine, and it is bound to be many times faster than the QDOS equivalent.

To plot points, we must first work out the screen address for a given pixel, given the X and Y coordinates. To make it easier, we shall assume that the screen is always 512 × 256 pixels, in either mode. Now, each line takes up 128 bytes, so for a given Y coordinate the start of the line is at location $20000+128*Y, as $20000 is the screen start address. As 512 pixels across correspond to 128 bytes, the number of bytes across the screen is X/4 (as 512/128=4). Thus, given X and Y the screen memory location is $20000+128*Y+X/4. (You can check this from BASIC if you

like.) As the screen is handled in words, it is important later on that the address is even. As this calculation is the same for both modes, let's see how this translates into 68008 code (assume X=D4.W, Y=D5.W):

LSR.W	#2,D4	divide X by 4
ANDI.L	#$007E,D4	make it 0–126 (and even)
MOVEA.L	#$20000,A0	screen start
ADDA.W	D4,A0	add (X/4)
LSL.W	#7,D5	multiply Y by 128
ADDA.W	D5,A0	and add that in so A0=screen start

After this, the screen address will be in register A0. Note the way the logical shifts were used to divide the coordinates by powers of two.

To plot pixels, you have to work out a new word based on the desired colour, read in the existing word from the screen, combine the two, and write it back to the screen. The full listing follows, with three main parts — HIPLOT, which plots in 4 colour mode, PIXCALC, which is similar to the code above, and LOPLOT, which plots in 8 colour mode. To use HI and LOPLOT, the x coordinate has to be in D1, the y coordinate in D2, and the colour in D3. These registers were chosen to make it easy to use with the CALL statement from BASIC.

Listing 7.1: Plotting Routines

```
                    4 colour mode plotting
                    D1=x, D2=y, D3=colour

6124        HIPLOT  BSR     PIXCALC             work out address
ED4B                LSL.W   #6,D3               D3=.......GR.000000
3803                MOVE.W  D3,D4               D4=.......GR.000000
EF4C                LSL.W   #7,D4               D4=GR.0000000000000
02448000            ANDI.W  #$8000,D4           D4=G000000000000000
02430080            ANDI.W  #$0080,D3           D3=00000000R0000000
8644                OR.W    D4,D3               D3=G0000000R0000000
02410007            ANDI.W  #7,D1               D1=no of rotates
343C7F7F            MOVE.W  #$7F7F,D2           D2=0111111101111111
E27A                ROR.W   D1,D2               rotate mask
E27B                ROR.W   D1,D3               rotate colour
C550                AND.W   D2,(A0)             D2=screen colour
8750                OR.W    D3,(A0)             put in new colour
4280                CLR.L   D0                  prepare D0
4E75                RTS                         then back to BASIC

                    Calculate screen address in A0
                    given x co-ord in D1, y co-ord in D2

3801        PIXCALC MOVE.W  D1,D4               D4=x co-ord
3A02                MOVE.W  D2,D5               D2=y co-ord
48C5                EXT.L   D5                  make y Long
E44C                LSR.W   #2,D4               /4 to get 'bytes across
02840000            ANDI.L  #$007E,D4           D4=0-126 even
007E
207C0002            MOVE.L  #$20000,A0          start of screen
```

```
0000
D0C4                ADDA.W D4,A0                add in x
EF4D                LSL.W  #7,D5                multiply y by 128
D0C5                ADDA.W D5,A0                and add in to result
4E75                RTS                         then exit

                    8 colour routine, same parameters
                    as HIPLOT

61E2      LOPLOT    BSR    PIXCALC              calculate address
ED4B                LSL.W  #6,D3                D3=.......RGB000000
3803                MOVE.W D3,D4                D4=D3
EF4C                LSL.W  #7,D4                D4=RGB0000000000000
02448000            ANDI.W #$8000,D4            D4=R0000000000000000
024300C0            ANDI.W #$00C0,D3            D3=00000000GB000000
8644                OR.W   D4,D3                D3=R0000000GB000000
02410006            AND.W  #6,D1                D1=0 to 6 even
343C7F3F            MOVE.W #$7F3F,D2            D2=0111111100111111
E27B                ROR.W  D1,D3                rotate colour
E27A                ROR.W  D1,D2                rotate mask
C550                AND.W  D2,(A0)              read screen
8750                OR.W   D3,(A0)              put in colour
4280                CLR.L  D0                   ready for BASIC
4E75                RTS                         then back
```

The way HIPLOT works is by firstly calling PIXCALC to work out A0, and then a series of shifts and ANDs are done to manipulate the colour value to correspond to something like **Figure 7.3**. Note that bit 0 of D3 is ignored, to correspond to the colour table previously. It is difficult to explain in words what each rotate does, so down the side I've shown the binary form of the registers to make it clearer. The bits marked G and R stand for green and red, and those marked '.' mean 'don't care'. After the OR, D3 contains a value corresponding to the correct screen format for the G and R bits. D2 is used as a mask, and they are both rotated the correct number of times to correspond to the x coordinate AND 7. D2 is then ANDed with the screen contents, to remove any colour in the pixel, then D3 ORed with the screen, to put the desired colour into the screen.

PIXCALC is not quite the same as above. Firstly, the values of D1 and D2 are transferred to D4 and D5 respectively, so their values do not get corrupted. Otherwise it is much the same as before.

LOPLOT is the 8 colour routine, and similar actions to HIPLOT are taken, but suitably altered to suit the screen format in **Figure 7.5**. Register D1 gets ANDed with 6 to make sure the number of rotates is 0, 2, 4, or 6 only, as it *must* be an even number. The state of the FLASH bits on the screen is left unaltered by it.

Printing characters

Characters on the screen consist of various numbers of plotted pixels, so the above routines could be extended. This has two disadvantages — the first is that the routine has to be written, which is quite tricky, and the

second is that the location of the character set varies on different versions of the QL ROM. In any case, there's not much point in reinventing the wheel — quite a bit of the ROM is concerned with character output via windows, so it is much easier to use a couple of QDOS traps. Without getting too involved in QDOS, basically all input/output relies on channel IDs. Note that these IDs are not the same as channel numbers in SuperBASIC, but are based on a similar principle.

Really the simplest way of putting characters on the screen is by using existing windows — the most obvious being the window that the BASIC PRINT statement sends data to. In BASIC this is channel #1, but to QDOS it is channel ID $00010001. The main output trap is TRAP #3, with the following parameters:

D0 byte — 5 to signal 'output'
D1 byte — character code to be printed
D3 word — timeout, usually −1
A0 long — channel ID

Generally speaking, when you want to send characters, you don't want any register values altered. Regrettably the trap does alter certain values, so to get round this here is a general routine called PRINT that sends character D1 to the screen window, without altering any register values.

Listing 7.2: PRINT Subroutine

```
                    prints chr$(D1) onto channel 1
                    without corrupting any registers
                    NOTE: output is lost if error in process

48E7D0C0 PRINT      MOVEM.L D0-1/D3/A0-1,-(A7)      save registers
207C0001            MOVE.L  #$00010001,A0           set channel ID
0001
76FF                MOVEQ   #-1,D3                  set 'timeout'
7005                MOVEQ   #5,D0                   signal 'output'
4E43                TRAP    #3                      do the printing
4CDF030B            MOVEM.L (A7)+,D0-1/D3/A0-1      restore regs
4E75                RTS                             then exit
```

It is quite straightforward — first the necessary register values are saved on the stack, then the required parameters selected, and the trap done. Finally all the register values are restored, and an RTS done. Beware though — if the window is not open, nothing will get printed, and the program will be none the wiser, and neither will you!

Suppose you don't want to use any of the standard windows for output? Well, the way to do it is to open a channel to suit, and the result of a

successful operation is that a new channel is created, with an exclusive ID number. The 'open trap' is #2, with these parameters:

D0 byte — 1 to signal 'open'
D1 long — job ID, normally −1
D3 long — 2 to signal 'new channel'
A0 long — address of channel name

A0 should point to the channel name — this should be in the form of a word showing the length of the channel name, followed by the ASCII of the name itself. (The channel name must start at an even address.) Registers D1–D3 and A1–A3 are corrupted by the call, D0 returns non-zero if any error has occurred, whereas if there are no errors then A0 returns with the channel ID (long).

When a new channel has been successfully opened, a routine similar to PRINT above can be used to send characters, but with suitable alteration to the value of A0.

The inverse function of open is, guess what, 'close'. This is TRAP #2, with two parameters:

D0 byte — 2 to signal 'close'
A0 long — channel ID of the channel to be closed

If D0 returns with a non-zero value, then it indicates that the channel was never open in the first place.

There is a lot more that can be done with these QDOS calls, but this is enough to send characters to the screen. The other most useful QDOS calls are for accessing the 8049 processor, in order to read the keyboard.

The 8049 second processor

There are two processors in the QL — the 68008 does all the hard work, like running SuperBASIC and QDOS, while at the same time a second processor, an Intel 8049, does the easy bits, like scanning the keyboard, controlling the serial ports, and other monotonous tasks. It takes care of the drudgery of looking after a computer such as the QL.

Unlike the 68008, the 8049 is not directly programmable, as its program area is not accessible, and is fixed in ROM anyway. It communicates with the 68008 by two ports, but the method is quite complex and you don't need to know how it works to use it. The 8049 is known as the IPC, which stands for 'independent peripheral controller'.

The keyboard

The keyboard is connected to the 8049 only, and the 68008 scans it by asking the 8049 to tell it what keys are being held down. The normal way of scanning is by interrupts, with the 8049 continually inserting the key pressed into a queue in the 68008's memory. The way to read the queue is by another couple of QDOS system traps, by asking for input from a 'con_' type of channel. The easiest to use is the same channel as we used for output — the one with an ID of $00010001. The relevant trap is TRAP #3, 'wait for a key to be pressed', which requires the following parameters:

D0 byte — 1 to signal 'fetch key'
D3 word — timeout, normally −1
A0 long — channel ID, normally $00010001
On return D1 (byte) contains the ASCII of the key pressed

There is another way to scan the keyboard, useful for multiple key presses, or if the usual buffering is not required, using the QDOS equivalent of the BASIC KEYROW function. This uses a trap to call the 8049 directly, by doing the following:

Listing 7.3: QDOS Version of KEYROW

```
                    requires row number in D1 (byte)
                    exits  with value in D1 byte

47FA0022            LEA     TEMP(PC),A3          A3=storage area
26BC0901            MOVE.L  #$09010000,(A3)      store parameters
0000
426B0004            CLR.W   4(A3)
17410006            MOVE.B  D1,6(A3)             store the parameter
177C0002            MOVE.B  #2,7(A3)
0007
7011                MOVEQ   #17,D0               signal IPC command
4E41                TRAP    #1                   use the 8049
4E75                RTS                          then exit
00000000 TEMP:      DS      8                    8 spare bytes for command
00000000
```

It uses TRAP #1 with D0 set to 17, which is the IPC communications trap. It requires A3 to point to a command, which has to be in the form of various parameters. There are 15 different commands, distinguished by the first byte, and in this case command 9 (keyboard direct read) is used. On exit from the routine D1 byte will contain the equivalent of the BASIC KEYROW function.

The alternate screen

The QL hardware allows two different areas to be used for the screen memory — $20000, which we have seen, and $28000. The memory map

for the alternate screen is identical to the normal one, but with $8000 added to all addresses. Unfortunately the QL firmware does not easily allow use of this 'screen #1', only the usual 'screen #0'.

The problem is caused by the fact that screen #1's memory is exactly where all the important system variables and tables lie. If you just switch it in and write to it, sooner or later the system will crash when you corrupt something important. From examining a listing of QDOS, it can be seen that originally the system variables could lie anywhere in RAM, addressed by register A6. Unfortunately at some stage this facility was disabled, so that QDOS is forced to have its variables at $28000. (Interestingly, the window channels still have as one of their parameters a long word that defines which screen they refer to.) Thus, to use screen #1, QDOS must lose all its variables, so it has to be disabled first.

Switching off QDOS

To disable QDOS, a certain sequence of events has to be followed. This is as follows:

(i) Go into supervisor mode, using TRAP #0. This is to stop QDOS trying to multi-task, which requires certain tables to be in RAM.
(ii) Disable interrupts, by ORI #$0700,SR. This is because the interrupt handler uses the system variables.
(iii) Make the stack pointer A7 a suitable value, as it may be in screen #1's memory area.

Having done this, the screens may be switched by directly writing to the register at $18063 described previously, with the value of bit 7 controlling the screen number.

With QDOS disabled, most TRAPs and system calls will crash the QL. The only safe one really is the trap that accesses the 8049 IPC, which is just as well as it is the only way to read the keyboard. Most other traps require some system variables to be present, which they won't be, so don't use them. It means of course that you'll have to write your own routines for printing characters and the like, and commands using I/O devices such as the microdrives are not possible. As the primary use for two screens is in games software, this is not a great limitation. To plot points, you could use HIPLOT and LOPLOT described previously, after changing the initial value of A0 from $20000 to $28000 for screen #1.

CHAPTER 8

An A–Z of the 68008 Instruction Set

This chapter contains all the information necessary to code all the 68008 instructions, with details and hints on each. It is best to work out the opcode itself in binary, and subsequent bytes for addressing in hex. Calculating the addressing bytes may seem at first to be very involved, but it is straightforward as long as you don't try to rush things.

Calculating addressing bytes

Every calculation that has an address field in it uses six bits to define the address. This is split into two 3 bit fields, one defining the mode, the other a register number. The allowed addressing modes are:

Address	Mode	Register
Dn	000	reg no.
An	001	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Both the mode and register are expressed in binary, for easy incorporation into each opcode. This is the complete list, but few instructions allow all addressing modes. For this reason, a form of the above table is included with each instruction opcode. Some types of addressing modes require extra words after the opcode.

Address & Data register direct An & Dn

Both these modes use the register field to hold the relevant register number, and require no extra bytes. For example, the instruction

CLR.L D2

would have as its address mode 001 and register field 010.

Address register indirect (An),(An)+,−(An)
All three modes use the register field to hold the relevant register number. None requires any extra bytes.

Address register indirect with displacement d(An)
This mode uses the register field to hold the register number, and requires one word of extension. The word contains the sign-extended displacement — for example, the instruction

NOT.W $1AA(A3)

is coded as follows. The addressing mode is 101, and the register field 011. This gives an opcode of 0100011001101011=$466B. The displacement has to be added afterwards, giving a final instruction of two words: 466B 01AA.

Address register indirect with index d(An,A/Dn.x)
This mode uses the register field to hold the address register number, and requires one additional word. The additional word consists of a byte containing the 8 bit displacement, and another byte holding other information. The format of this word is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	r	r	r	W/L	0	0	0	d	d	d	d	d	d	d	d

Bit 15 defines the index register — 0 for data, 1 for address.
Bits 14–12 define the index register number.
Bit 11 defines the size of the index register — 0 for W, 1 for L.
Bits 10 to 8 should be zero.
Bits 7 to 0 determine the 8 bit sign-extended displacement.

For example, the instruction

CMP.L $23(A3,D5.W),D0

has an addressing mode of 110, and register field of 011 (from the first register). The opcode is

1011000010110011=$B0B3

The extension word becomes

01010000 00100011=$5023

forming the full instruction

B0B3 5023

Absolute word nn.W
This has an addressing mode of 111, and a register field of 000. It requires one word of extension — the absolute word itself. For example, the instruction

JSR $3574.W

has an opcode of

0100111010111000=$4EB8

and has an extension of the word itself, ie $3574 giving a full instruction of

4EB8 3574

Absolute long nn.L
This has an addressing mode of 111, and a register field of 001. It requires a long word of extension, being the address itself. For example, the instruction

JMP $3FC00

would have an opcode of

0100111011111001=$4EF9

and a long word extension of $0003FC00, giving a full instruction of

4EF9 0003 FC00

Program counter with displacement n(PC)
This has an addressing mode of 111, with a register field of 010. It requires one word of extension, the word being the sign-extended displacement. The displacement is calculated by subtracting the *location of the extra word* from the *absolute value of the destination*. For example, if at location $3F000 there is an instruction

LEA $3FC00(PC),A1

the first thing is to work out the opcode, which is 0100001111111010=$43FA which would go into location $3F000. The extension word is going into $3F002, and it calculates it to be $3FC00–$3F002, which is $0BFE, giving a full instruction of

43FA 0BFE

If you calculate the extension word to be an odd address, unless you are referencing byte data it probably means you have made a mistake somewhere.

Program counter with index n(PC,A/Dn.x)
This mode has an addressing mode of 111, and register field 011. It requires one word of extension, of a similar format as that for address register indirect with index thus:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	r	r	r	W/L	0	0	0	d	d	d	d	d	d	d	d

In this case though, the 8 bit sign-extended displacement is calculated in the same way as for PC mode, namely by subtracting the location of the extra word from the absolute location. Note that the limiting values of the displacement are from −128 to 127. If you get a displacement of over 127, you'll have to reorganise the code, using LEA, by replacing something like

MOVE.W TOOFAR(PC,A0.L),D1

with

LEA TOOFAR(PC),A1
MOVE.W 0(A1,A0.L),D1

As an example, if at location $3FC00 you had the instruction

CMP.B $3FC40(PC,D0.W),D1

its opcode would be

1011001000111011=$B23B

and the extra word would be at location $3FC02, giving a displacement of $3FC40−$3FC02=$3E, and an extra word of

0000000000111110=$003E

giving a full instruction of

B23B 003E

Immediate data #nn
This has an addressing mode of 111 with a register field of 100, and requires one or two extra words, depending on the size of the instruction itself. For byte-sized instructions, the extra word has its most significant byte as zero, with the data in the least significant byte. Word-sized instructions have as the extra word the data, while long-sized instructions have two extra words — firstly the most significant word, then the least significant word. For example, the instruction

MOVEA.L #$28000,A6

has an opcode of

0010110001111100=$2C7C

and two extra words for the data, giving a full instruction of

2C7C 0002 8000

Mnemonics

In the following pages, each instruction is given a general mnemonic. It does not contain any indication of the size of the instruction, and to be as general as possible uses certain abbreviations:

(address)	Signifies addressing mode, the permitted modes being shown in the table.
x	Used for register numbers, and ranges from 0–7. If two registers are involved in the instruction, 'y' may also be used as a register number.
(d)	Used for data, range usually depending on the instruction size.
(location)	Denotes a location or more usually a program label.
(reg list)	Used to signify a register list, such as D0–3/A3/A6.

Size

Some instructions have only one size, which need not be specified in the mnemonic. Other instructions can have various sizes which need to be

specified in the mnemonic, while for the rest, size has no meaning, denoted by n/a.

Condition codes

Shown for each instruction is the result on each condition code flag. The following notation is used:

0 – flag reset
1 — flag set
* — flag is altered
– — flag is not affected
? — flag may be either state — unpredictable

Where the state of the flag does not follow the norm, it will be explained underneath.

Opcode

This is expressed in binary, with certain bits being denoted by abbreviations. Each abbreviation is defined underneath.

Addressing modes

The table indicates the allowed addressing modes for each instruction, along with the mode and register numbers in binary.

ABCD — ADD BINARY CODED DECIMAL

Mnemonic: ABCD Dx,Dy and ABCD −(Ax),−(Ay)

Size: byte only

Action: the BCD byte specified first is added to the BCD byte specified second, and the state of the extend bit also added. In the first form, both bytes are the contents of the lowest eight bits of the data registers, while in the second form they lie at locations Ax−1 and Ay−1, and the values of the address registers are decremented.

Condition codes:

X	N	Z	V	C
*	?	*	?	*

Z — cleared if the result is non-zero, else *unchanged*
C — set if a decimal carry occurred, ie sum>99
X — same as carry flag

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	y	y	y	1	0	0	0	0	R/M	x	x	x

y — second (destination) register number
x — first register number
R/M — 0 — data register direct addressing used
1 — address register pre-decrement addressing used

Notes: BCD data is stored in bytes, one byte taking two digits. For example, the number 27 decimal is stored as $27.

ADD — BINARY ADD

Mnemonic: ADD (address),Dx and ADD Dx,(address)

Size: byte, word or long

Action: the first parameter is added to the second parameter. Only the bits specified by the size are used and altered.

Condition code:

X	N	Z	V	C
*	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	x	x	x	om	s	s	m	m	m	r	r	r

x — data register number
om — op-mode field: 0 — ADD (address),Dx
1 — ADD Dx,(address)
s — size: 00 — byte
01 — word
10 — long

Addressing modes:

Mode	mmm	rrr	ADD (address),Dx	ADD Dx,(address)
Dn	000	reg no.	Y	N
An	001	reg no.	Y (not byte)	N (use ADDA)
(An)	101	reg no.	Y	Y
(An)+	011	reg no.	Y	Y
−(An)	100	reg no.	Y	Y
n(An)	101	reg no.	Y	Y
n(An,A/Dn)	110	reg no.	Y	Y
nn.W	111	000	Y	Y
nn.L	111	001	Y	Y
n(PC)	111	010	Y	N
n(PC,A/Dn)	111	011	Y	N
#nn	111	100	Y	N

Notes: there are other forms of binary add — ADDA, when the destination is an address register, ADDI, when immediate data has to be added to an address, and ADDQ for fast addition.

ADDA — ADD ADDRESS

Mnemonics: ADDA (address),Ax

Size: word or long — not byte

Action: the address is added to the contents of the specified address register. For word operations, the sum is sign-extended to 32 bits before being placed in the address register.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	x	x	x	s	1	1	m	m	m	r	r	r

x — address register number
s — size, 0 — word, 1 — long
m — address mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
An	001	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: the fact that none of the condition codes are affected by this can be a great source of problems when debugging, so beware. Most assemblers should automatically distinguish between this and the normal ADD.

ADDI — ADD IMMEDIATE

Mnemonic: ADDI #(d),(address)

Size: byte, word or long

Action: the immediate data is added to the specified address. The size of the data corresponds to the instruction size.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	s	s	m	m	m	r	r	r

followed by one or two extra words containing the data. For byte-sized operations, one extra word is needed, the data contained in the least significant byte. Word-sized operations have the data in a single extra word, and long operations have two extra words. Additional addressing bytes come after these additional words.

s — size, 00 byte, 01 word, 10 long

m — addressing mode

r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: it is illegal to try ADDI #(d),Ax — use ADDA #nn,Ax or ADDQ #nn,Ax instead. For immediate data in the range 1–8 it is faster and more economical to use ADDQ.

ADDQ —ADD QUICK

Mnemonic: ADDQ #(d),(address)

Size: byte, word or long

Action: as for ADDI, but with a very limited data range of 1–8

inclusive. The advantage is that it takes less bytes, and executes around twice as fast.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Important: the condition codes are not affected by ADDQ to an address register.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	d	d	d	0	s	s	m	m	m	r	r	r

d — immediate data. The same as the data in the instruction, unless #8 is specified, when the data in the instruction is 0.

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
An	001	reg no. (not byte)
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: normally only the relevant part of the address is altered by this instruction, depending on the size, except with the address register — with it, the result is sign-extended to long for word operations.

ADDX — ADD WITH EXTEND

Mnemonic: ADDX Dx,Dy or ADDX −(Ax),−(Ay)

Size: byte, word or long

Action: the first parameter is added together with the second and the state of the extend flag, and the result put back in the second parameter. The parameters can be specified either by direct data addressing, or pre-decrement indirect addressing.

Condition codes:

X	N	Z	C	V
*	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	y	y	y	1	s	s	0	0	r/m	x	x	x

y — first register number
s — size, 00 byte, 01 word, 10 long
r/m — 0 – Dx,Dy, 1 – –(Ax),–(Ay)
x — second register number

Notes: this is mainly used for multiple precision maths operations.

AND — LOGICAL AND

Mnemonic: AND (address),Dx and AND Dx,(address)

Size: byte, word or long

Action: the first parameter is logically ANDed with the second, and the result placed back in the second. Only the specified part of both parameters are read and altered.

Condition codes:

X	N	Z	C	V
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	x	x	x	om	s	s	m	m	m	r	r	r

x — data register number
om — 0 for AND (address),Dx: 1 for AND Dx,(address)
s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr	AND (address),Dx	Dx,(address)
Dn	000	reg no.	Y	N
(An)	010	reg no.	Y	Y
(An)+	011	reg no.	Y	Y
–(An)	100	reg no.	Y	Y
n(An)	101	reg no.	Y	Y
n(An,A/Dn)	110	reg no.	Y	Y
nn.W	111	000	Y	Y
nn.L	111	001	Y	Y
n(PC)	111	010	Y	N
n(PC,A/Dn)	111	011	Y	Y
#nn	111	100	Y	N

Notes: it is not possible to have an address register as either parameter in an AND instruction. It is necessary to MOVE the address register to a data register, and use the data register as the parameter. There are three

other forms of AND — ANDI, for immediate data, and two ANDs for modifying the status register.

ANDI — LOGICAL AND IMMEDIATE

Mnemonic: ANDI #(d),(address)

Size: byte, word or long

Action: the immediate data is logically ANDed with the address, and the result placed back in the address.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	s	s	m	m	m	r	r	r

followed by one or two extra words to hold the data. For byte operations, the least significant byte of the extra word is used for the data, word operations, require an extra word, and long operations require an extra two words.

s — size, 00 byte, 01 word, 10 long

m — addressing mode

r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
–(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: as with AND, it is not possible to specify an address register in the instruction.

ANDI TO SR/CCR — AND IMMEDIATE TO STATUS REGISTER

Mnemonics: ANDI #(d),CCR and ANDI #(d),SR

Size: byte for CCR, word for SR

Action: the immediate data is ANDed with the CCR or SR,

and the result placed back into it. Note that with the SR *the instruction is privileged.*

Condition codes:

X	N	Z	V	C
*	*	*	*	*

the value of each flag will be cleared or left alone, depending on the state of the relevant bit in the immediate data.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	x	1	1	1	1	0	0

followed by an additional word. The word contains the immediate data (for ANDI to SR), or the low byte contains the immediate data (for ANDI to CCR).
x — 0 for CCR (byte), 1 for SR (word)

Notes: ANDI to SR is privileged as it lets the programmer alter the supervisor byte of the SR. Common instructions and their opcodes are:

```
027C DFFF  ANDI #$DFFF,SR  go into user mode
027C DFFF  ANDI #$7FFF,SR  trace off
023C 0000  ANDI #$00,CCR   clear all flags
```

ASL/R — ARITHMETIC SHIFT LEFT & RIGHT

Mnemonics: ASL Dx,Dy and ASL #(d),Dx and ASL (address) and ASR Dx,Dy and ASR #(d),Dx and ASR (address)

Size: byte, word or long (word only for (address))

Action: an arithmetic shift is one in which the operand has a number of bits shifted to the left or right, best illustrated by a diagram (see **Figure 8.1**). The number of shifts and the item shifted can be specified in a number of ways: using Dx,Dy register Dy is shifted by the value of Dx, MOD 64. Using #(d),Dx register Dx is shifted a number of times depending on the immediate data, from 1 to 8. Using just (address), the contents of the address are shifted once only, and the size is always word. Only the relevant part of the parameter is shifted, depending on the size.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

X — set according to the diagram, or unaffected by a zero shift count.

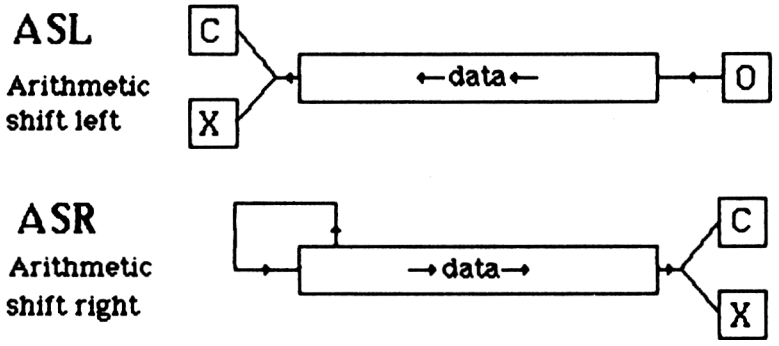


Figure 8.1: Arithmetic Shifts.

N — set if the highest bit of the result is set.
Z — set if the result is zero, else cleared.
V — set if the highest bit changes at any time, else cleared.
C — set according to the diagram, or unaffected by a zero shift count.

Opcodes:

for Dx,Dy:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	x	x	x	l/r	s	s	1	0	0	y	y	y

x — first register (count register) number
l/r — direction, 0 for right, 1 for left
s — size, 00 byte, 01 word, 10 long
y — second register number (the one shifted)

for #(d),Dx:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	d	d	d	l/r	s	s	0	0	0	x	x	x

d — immediate data, from 0–7, 0 gives a shift count of 8
l/r — direction 0 for right, 1 for left
s — size, 00 byte, 01 word, 10 long
x — data register number

for (address) (ie memory shifts):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	l/r	1	1	m	m	m	r	r	r

l/r — direction, 0 for right, 1 for left
m — addressing mode
r — address register

Addressing modes (for memory shifts only):

Mode	mmm	rrr
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

BCC — BRANCH ON CONDITION

Mnemonic: Bcc (location)

Size: byte or word

Action: if the specified condition is true, then a branch is made to the desired location. If not, the instruction after the branch is executed. The destination of the branch is not stored as an absolute number, but as a two's-complemented displacement, making the instruction position independent.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	c	c	c	c	d	d	d	d	d	d	d	d

which may be followed by an extra word.

c — condition, one of:

0010 HI
0011 LS
0100 CC
0101 CS
0110 NE
0111 EQ
1000 VC
1001 VS
1010 PL
1011 MI
1100 GE
1101 LT
1110 GT
1111 LE

d — the displacement. If this is zero, then an extra word is used for a 16 bit displacement. In either case,

the displacement is calculated by subtracting the *(location of the Branch opcode+2) from the absolute value of the destination*. If the result is less than 128 then an 8 bit displacement can be used. If it is from −2 to −128 then the two's-complement can be used as an 8 bit displacement. If the result does not fall into either of these ranges, then the 8 bit displacement has to be zero, and an extra 16 bit signed displacement added. For example, if at location $3FC00 there are two instructions
BEQ $3FC50
BPL $3FF00
the respective displacements would be 3FC50−3FC02 =004E and 3FF00−3FC04=02FC, giving the opcodes of
674E
6A00 02FC

Notes: the two 'missing' condition codes correspond to these instructions:
0000 — BRA, unconditional branch
0001 — BSR, branch to subroutine
If you manage to calculate an odd displacement then you've made a mistake somewhere.

BTST, BCHG, BCLR, BSET — TEST BIT INSTRUCTIONS

Mnemonics: BTST Dx,(address) and BTST #(d),(address)
BCHG Dx,(address) and BCHG #(d),(address)
BCLR Dx,(address) and BCLR #(d),(address)
BSET Dx,(address) and BSET #(d),(address)

Size: byte (for memory) and long (for data registers)

Action: these are a family of instructions of the form 'Test a bit and ?'. They all test a particular bit of an address, then alter the bit in some way.

BTST — test a bit (and do nothing with it)
BCHG — test a bit and change — the bit gets complemented
BCLR — test a bit and clear — the bit gets reset
BSET — test a bit and set — the bit gets set to 1

The bit number can be expressed in two ways — either as immediate data, using #(d), or by having the bit

number in a data register. If (address) is a data register, then the bit number is reduced MOD 32 allowing access to all its bits. If (address) is not a data register, then the memory is referenced as byte-sized, and the bit number taken MOD 8.

Condition codes:

X	N	Z	V	C
–	–	*	–	–

Z — set if the bit tested was zero, else reset

Opcodes:

there are two forms of opcodes — for dynamic bit number, in a data register, or static bit number, as immediate data.

Dynamic bit number — Dx,(address)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	x	x	x	1	c	c	m	m	m	r	r	r

x — data register number
c — instruction type:
BTST — 00
BCHG — 01
BCLR — 10
BSET — 11
m — addressing mode
r — address register

Immediate data — #(d),(address)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	c	c	m	m	m	r	r	r

followed by an extra word, containing the immediate data.
c — instruction type (see above)
m — addressing mode
r — address register

Addressing modes:

The same addressing modes are allowed in all forms of the instruction, namely:

Mode	mmm	rrr
Dn	000	reg no. (long sized)
(An)	010	reg no.
(An)+	011	reg no.
–(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: To test bit numbers greater than 7 on memory addresses, MOVE the memory contents into a data register of suitable size, and use the static bit number form of the instruction. Note that an address register can not be directly specified in any of these instructions.

BRA — BRANCH ALWAYS

Mnemonic: BRA (location)

Size: byte or word

Action: this is an unconditional branch instruction, and the destination is specified relative to the instruction.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	d	d	d	d	d	d	d	d

may be followed by an extra word.
d — 8 bit displacement — if this is zero then an extra word follows, giving a 16 bit sign extended displacement. Both values are worked out in the same way as the Bcc instructions, namely by subtracting the location of the Branch opcode+2 from the absolute value of the destination.

Notes: it is not possible to have a short branch to the following opcode, only a long branch. This is useful to remember when debugging by hand.

BSR — BRANCH TO SUBROUTINE

Mnemonic: BSR (location)

Size: byte or word

Action: the address of the instruction following this one is placed on the stack, then control passes to the desired location. It stores the destination in the same relative form as the branch instructions.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1	d	d	d	d	d	d	d	d

d — 8 bit two's-complemented displacement
may be followed by an extra word if d=0. The value and size of the displacement is calculated in the usual way (see Bcc and BRA).

Notes: this is a relative form of the JSR instruction, but without the flexibility of the addressing modes of JSR.

CHK — CHECK REGISTER

Mnemonic: CHK (address),Dx

Size: word only

Action: the lowest 16 bits of the data register are compared to the two's-complement value at (address). If the register value is less than 0 or greater than the specified value, a CHK exception will occur. This is to allow fast checking of boundary limits on such things as arrays.

Condition codes:

X	N	Z	V	C
–	*	?	?	?

N — set if Dx<0, reset if Dx is greater than (address), else undefined

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	x	x	x	1	1	0	m	m	m	r	r	r

x — denotes data register number
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
DN	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
–(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: all addressing modes are allowed, with the exception of address register direct. As the CHK exception normally does nothing on the QL, this is not really very useful. However, it is one of the RAM re-definable vectors, so it could be used in your own programs for limit checking.

CLR — CLEAR

Mnemonic: CLR (address)

Size: byte, word or long

Action: the specified address is zeroed. The number of bits set to 0 depends on the specified size of the instruction. It is the fastest way of putting 0 into memory or registers.

Condition codes:

X	N	Z	V	C
–	0	1	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
–(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: this cannot be used to zero address registers, nor can it be memory addressed by either of the PC modes. Probably the most popular use on the QL is zeroing D0, ready for returning to BASIC. The instruction for this is CLR.L D0 which has an opcode of $4280.

CMP — COMPARE

Mnemonic: CMP (address),Dx

Size: byte, word or long

Action: the address is subtracted from the specified data register, the condition codes being suitably altered. The actual result of the subtraction is not stored.

Condition codes:

X	N	Z	V	C
–	*	*	*	*

C — set if a borrow occurs (eg 1–5)

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	x	x	x	0	s	s	m	m	m	r	r	r

x — data register number
s — size, 00 byte, 01 word, 10 long

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
An	001	reg no. (not byte)
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: all addressing modes are allowed. There are three other forms of CMP — CMPA, for comparison with address registers, CMPI, for immediate data, and CMPM for comparing memory. Most assemblers should automatically select the correct type, but beware when hand coding.

CMPA — COMPARE ADDRESS

Mnemonic: CMPA (address),Ax

Size: word or long

Action: the contents of (address) are subtracted from the specified address register, and the condition codes suitably affected. The result of the subtraction is not stored. If the size is word then both parameters are sign-extended to 32 bits before the subtraction.

Condition codes:

X	N	Z	V	C
–	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	x	x	x	s	1	1	m	m	m	r	r	r

x — address register number
s — size, 0 word, 1 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
An	001	reg no.
(An)	010	reg no.
(An)+	011	reg no.
–(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: all addressing modes are allowed.

CMPI — COMPARE IMMEDIATE

Mnemonic: CMPI#(d),(address)

Size: byte, word or long

Action: the immediate data is subtracted from the specified address, and the condition codes altered. As with all other compares, the result is not stored.

Condition codes:

X	N	Z	V	C
–	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

This has to be followed by one or two extra words, holding the immediate data. For byte operations, the

data is held in the lower byte of an extra word, word operations use an extra word, and long operations use two extra words to hold the data.

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

CMPM — COMPARE MEMORY

Mnemonic: CMPM (Ax)+,(Ay)+

Size: byte, word or long

Action: using post increment addressing, two parts of memory are compared, by subtracting the first from the second, altering the condition codes. The result is not stored. Because of the addressing mode used, the value of both address registers is incremented by 1, 2 or 4, depending on the size.

Condition codes:

X	N	Z	V	C
–	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	x	x	x	1	s	s	0	0	1	y	y	y

x — first address register
s — size, 00 byte, 01 word, 10 long
y — second address register

Notes: this instruction, used in conjunction with a DBEQ loop is very useful for searching for sequences in memory, particularly when tokenising program lines.

DBCC — DECREMENT AND BRANCH UNTIL CONDITION

Mnemonic: DBcc Dx,(location)

Size: word

Action: 'cc' in the mnemonic can be any of the 16 usual conditions. What happens when this instruction executes is: if the condition is true, then control passes to the following instruction; if not, the lowest 16 bits of the data register are decremented by one, and the branch made if the register does not equal −1. It is the main way of forming loops on the 68008, and the most common form is DBF, 'decrement and branch until false'. As the condition, by definition, will never be met, the loop will always execute until the data register reaches −1.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	c	c	c	c	1	1	0	0	1	x	x	x

followed by one extra word, for the displacement.

c — denotes condition, one of:

0000	T (no practical use in this instruction)
0001	F – most used form
0010	HI
0011	LS
0100	CC
0101	CS
0110	NE
0111	EQ
1000	VC
1001	VS
1010	PL
1011	MI
1100	GE
1101	LT
1110	GT
1111	LE

x — data register number

The extra word should contain the displacement, which is calculated by subtracting *the location of the extra word from the absolute destination*, with a range of $7FFF to −$8000 inclusive.

Notes: an alternate form of DBF is DBRA. When setting the initial value of the loop count register, you should remember that the loop will execute a maximum of the original value *plus one*, as it terminates on reaching −1, not 0 as would normally be expected.

DIVS — SIGNED DIVISION

Mnemonic: DIVS (address),Dx

Size: word only

Action: the contents of the specified data register are divided by the given address, and the result placed back in the data register. The original parameters are taken to be 16 bit quantities, and the result takes 32 bits — the lower 16 bits take the integer result of the division (the quotient), while the higher 16 bits hold the remainder. The division is calculated using signed arithmetic, and an attempt to divide by zero will cause the relevant exception to occur.

Condition codes:

X	N	Z	V	C
–	*	*	*	0

N — set if the integer result is negative
Z — set if the integer result is zero
V — set if the integer result is greater than 16 bits. If it does occur, the state of the N and Z flags will be undefined.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	x	x	x	1	1	1	m	m	m	r	r	r

x — data register number
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: this is one of four instructions that take varying amounts of time to execute, depending on the values of its parameters. On the QL, the divide by zero exception is usually ignored.

DIVU — UNSIGNED DIVISION

Mnemonic: DIVU (address),Dx

Size: word

Action: the contents of the data register are divided by the specified address, and the result placed back in the data register. Both original operands are taken as 16 bit, and the result is 32 bit — the lower 16 bits the integer result, the higher 16 bits the remainder. Unsigned arithmetic is used for the calculation. If an attempt is made to divide by zero, an exception will occur.

Condition codes:

X	N	Z	V	C
–	*	*	*	0

N — set if the integer result is negative
Z — set if the integer result is zero
V — set if the integer result cannot be contained in 16 bits. If an overflow does occur, the state of the N and Z bits is undefined.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	x	x	x	0	1	1	m	m	m	r	r	r

x — data register number
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: this instruction also takes a varying amount of time to execute.

EOR — EXCLUSIVE OR

Mnemonic: EOR Dx,(address)

Size: byte, word or long

Action: the contents of the data register are Exclusive ORed with the contents of the address, and the result placed in the address.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	x	x	x	1	s	s	m	m	m	r	r	r

x — data register number
s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
–(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: there are three other types of EOR — EORI, for immediate data, and two instructions for altering the status register.

EORI — EXCLUSIVE OR IMMEDIATE

Mnemonic: EORI #(d),(address)

Size: byte, word or long

Action: the immediate data is exclusive ORed with the address, and the result placed back into the address. The size of the immediate data is the same as the instruction size.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

The opcode should be followed by one or two extra words of data, depending on the instruction size.

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

EORI TO CCR/SR — EXCLUSIVE OR IMMEDIATE TO STATUS REGISTER

Mnemonic: EORI #(d),CCR and EORI #(d),SR

Size: byte for CCR, word for SR

Action: the immediate data is exclusive ORed with part or all of the status register. Because of the actions, *EORI to SR is a privileged instruction.*

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The state of the condition codes depend on the state of the relevant bits of the immediate data.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	x	1	1	1	1	0	0

followed by an extra word defining the immediate data.
x — 0 for CCR (byte), 1 for SR (word)

EXG — EXCHANGE REGISTERS

Mnemonic: EXG D/Ax,D/Ay

Size: long only

Action: the values of the two specified registers are exchanged. All 32 bits of each register are used. Data registers and address registers may be interchanged with themselves and each other.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	x	x	x	1	m	m	m	m	m	y	y	y

x — register number
m — register types:
01000 — Dx,Dy
01001 — Ax,Ay
10001 — Dx,Dy
y — register number

Notes: if a data register and address register are to be exchanged, the number of the data register is always x, and the address register is y. Do not confuse this with SWAP, or mis-type it as EXT, which are quite different instructions.

EXT — EXTEND

Mnemonic: EXT Dx

Size: word or long

Action: the data register is sign-extended to 16 or 32 bits, depending on the size. If the size is word, bit 7 of the data register is copied to bits 8 through 15, else if it is long then bit 15 is copied to bits 16 through 31.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	s	0	0	0	x	x	x

s — size, 0 word, 1 long
x — register number

Notes: beware that to extend a byte value to a long value, two instructions are needed:

```
EXT.W Dx
EXT.L Dx
```

ILLEGAL — ILLEGAL INSTRUCTION

Mnemonic: ILLEGAL

Size: n/a

Action: an illegal exception occurs, with the value on the supervisor stack being the location of this instruction.

There are many other instructions which can cause the exception, but some are used as valid instructions in other 68000 series processors. This opcode is guaranteed to remain illegal on all 68000 series processors.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0	=$4AFC

Notes: it may seem strange to guarantee the illegality of an instruction, but it can be a useful way of creating your own instructions by writing a suitable exception handler. This is particularly valid on the QL as the normal way of adding instructions, via line 1010 and 1111 opcodes, is not possible. The illegal exception vector is re-definable via RAM.

JMP — JUMP

Mnemonic: JMP (address)

Size: n/a

Action: control passes to the specified address.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	m	m	m	r	r	r

m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
(An)	010	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011

Notes: it is very useful that the destination of the jump can be defined by an addressing mode, particularly as it makes it easy to implement jump tables in memory. Be careful when working out the addressing modes for this

instruction (and JSR to follow). The address is calculated in the same way as LEA, and not the way MOVE does it.

JSR — JUMP TO SUBROUTINE

Mnemonic: JSR (address)

Size: n/a

Action: the address of the instruction following the JSR is put on the stack, then a jump made to (address). The address is specified as an addressing mode.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	0	m	m	m	r	r	r

m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
(An)	010	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011

Notes: as with JMP, the address is calculated in the same way as LEA, not MOVE.

LEA — LOAD EFFECTIVE ADDRESS

Mnemonic: LEA (address),Ax

Size: long only

Action: the address calculated from the addressing mode is placed into the address register. There is a subtle difference between this and the MOVE instruction. MOVE (and most others) calculates the address, then reads that memory location. LEA just does the calculating, covered in more detail in Chapter 5.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Addressing modes:

Mode	mmm	rrr
(An)	010	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011

Notes: probably the second most popular instruction when programming the QL. In particular, using it gets around the problem that memory cannot be written to directly using the PC modes: using LEA and an additional address register gets around the problem eg

```
LEA VARS(PC),A1
MOVE.L D1,(A1)
```

to replace the illegal

```
MOVE.L D1,VARS(PC)
```

LINK — LINK

Mnemonic: LINK Ax,#(d)

Size: n/a

Action: firstly the specified address register is pushed on the stack. The address register is then loaded with the new value of A7, then the sign-extended value of the 16 bit data is added to the address register. Normally the data will be negative, to make room on the stack (as the stack is upside down).

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	x	x	x

followed by an extra word containing the data.
x — address register number

Notes: the usual use for LINK is to reserve space on the stack for local variables and parameter passing. It has a complementary instruction UNLK. The use of these

commands is quite advanced, and will not be covered here.

LSL/R — LOGICAL SHIFT LEFT AND RIGHT

Mnemonics: LSL Dx,Dy and LSL #(d),Dx and LSL (address) and LSR Dx,Dy and LSR #(d),Dx and LSR (address)

Size: byte, word or long (word only for (address))

Action: a logical shift is one in which the operand has a number of bits shifted to the left or right, best illustrated by a diagram (see **Figure 8.2**). The number of shifts and the item shifted can be specified in a number of ways. Using Dx,Dy, register Dy is shifted by the value of Dx, modulus 64. Using #(d),Dx, register Dx is shifted a number of times depending on the immediate data, from 1 to 8. Using just (address), the contents of the address are shifted once only, and the size is always word. Only the relevant part of the parameter is shifted, depending on the size.

Condition codes:

X	N	Z	V	C
*	*	*	0	*

X — set according to the diagram, or unaffected by a zero shift count
N — set if the highest bit of the result is set
Z — set if the result is zero, else cleared
C — set according to **Figure 8.2** or cleared by a zero shift count

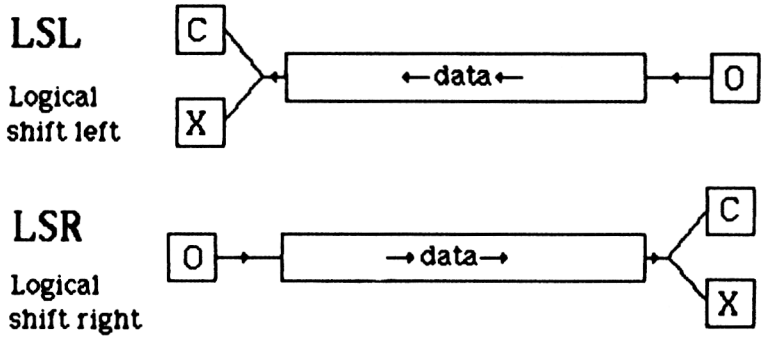


Figure 8.2: Logical Shifts.

Opcodes:

for Dx,Dy:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	x	x	x	l/r	s	s	1	0	1	y	y	y

x — first register (count register) number
l/r — direction, 0 for right, 1 for left
s — size, 00 byte, 01 word, 10 long
y — second register number (the one shifted)

for #(d),Dx:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	d	d	d	l/r	s	s	0	0	1	x	x	x

d — immediate data, from 0–7, 0 gives a shift count of 8
l/r — direction 0 for right, 1 for left
s — size, 00 byte, 01 word, 10 long
x — data register number

for (address) (ie memory shifts):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	l/r	1	1	m	m	m	r	r	r

l/r — direction, 0 for right, 1 for left
m — addressing mode
r — address register

Addressing modes : (for memory shifts only)

Mode	mmm	rrr
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

MOVE — MOVE DATA

Mnemonic: MOVE (source address),(destination address)

Size: byte, word or long

Action: the main instruction in the 68000 series, it transfers the contents of the source address into the destination address. Only the required part is transferred, depending on the size of the instruction.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

N — set if the data moved was negative
Z — set if the data was zero

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	s	s	r2	r2	r2	m2	m2	m2	m	m	m	r	r	r

s — size, 00 byte, 11 word, 10 long
r2 — destination address register
m2 — destination addressing mode
m — source addressing mode
r — source address register

The destination register and mode are specified in reverse of the usual order. To calculate an instructions opcode, first add extra bytes for the source addressing mode, then any extra destination addressing mode bytes.

Addressing modes:

Source:

Mode	mmm	rrr
Dn	000	reg no.
An	001	reg no. (not byte)
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Destination:

Mode	mmm	rrr
Dn	000	reg no.
An	see MOVEA	
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n (An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: note the non-standard way the size of MOVEs is determined, and the reverse order for the destination address fields. There are various other types of MOVE — MOVEA, for address registers, MOVEM, for multiple registers, MOVEP, for peripherals, and

MOVEQ, for fast data registers, as well as status register and stack pointer MOVEs.

MOVEA — MOVE ADDRESS

Mnemonic: MOVEA (address),Ax

Size: word or long

Action: the contents of the specified address are moved into the address register. For word-sized operations, the data is sign-extended to 32 bits before going into the address register.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

No condition codes are affected, unlike the normal MOVE instruction.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	s	x	x	x	0	0	1	m	m	m	r	r	r

s — size, 1 word, 0 long
x — address register number
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
An	001	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: all addressing modes are allowed, but byte operations are not possible. Most assemblers should automatically distinguish between this and MOVE.

MOVE TO SR/CCR — MOVE DATA TO STATUS REGISTER

Mnemonic: MOVE (address),CCR and MOVE (address),SR

Size: word (*including CCR*)

Action: the contents of the address are placed in part or all of the status register. For CCR, only the lower byte is used, while SR uses the whole word, and is thus a privileged instruction.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The state of the condition codes depends on the data moved to the CCR and SR.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	x	0	1	1	m	m	m	r	r	r

x — 0 for CCR, 1 for SR
m — addressing mode
r — address register

Notes: MOVE to CCR and MOVE from CCR are the only instructions using CCR to be word in size — all others are byte-sized.

MOVE FROM SR — MOVE DATA FROM STATUS REGISTER

Mnemonic: MOVE SR,(address)

Size: word

Action: all of the status register is moved into the specified address. It is particularly useful for saving the condition codes on the stack.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	m	m	m	r	r	r

m — addressing mode
r — address register

Addressing mode:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: the 68008 does not have a MOVE from CCR instruction, which is not a problem as this is not a privileged instruction. However, it is privileged on the 68010, so MOVE from CCR exists on that processor, with the same opcode as this, except that bit 9 is set. The most common form is used for saving the condition codes on the stack, namely MOVE SR,−(A7) which has an opcode of $40E7.

MOVE USP — MOVE USER STACK POINTER

Mnemonic: MOVE USR,Ax and MOVE Ax,USP

Size: long only

Action: the user stack pointer can be read and altered with this instruction, using another address register. This is a privileged instruction, designed for programs running in supervisor mode to access and alter the user stack pointer.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	c	x	x	x

c — 0 for Ax,USP, 1 for USP,Ax

x — address register number

MOVEM — MOVE MULTIPLE

Mnemonic: MOVEM (address),(registers) and MOVEM (registers),(address)

Size: word or long

Action: the specified registers are moved to or from memory. The registers are denoted by a list, such as D0–3/A0/A3, 'adjacent' registers separated by '–'s, and others separated by '/'. The order of the registers in the list does not matter, as it is determined by the processor.

In memory to register operations, the memory contents are transferred into registers starting at the address and working up to higher addresses, in the order D0 to D7, then A0 to A7. If the source is specified using post-increment addressing, the final value of the address register will be the address of the last register loaded plus two. For word operations the value of each register is sign-extended to 32 bits.

In register to memory commands, the action depends on the addressing mode used. For most of them, the register values get transferred to memory starting at the given address and working up, in the same order as above. The exception is pre-decrement addressing. With that, the registers get transferred starting at the given address *minus two*, and then downwards. The final value of the address register is the address of the final word stored. The order is the reverse to that given, namely A7 to A0, then D7 to D0.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	M/R	0	0	1	s	m	m	m	r	r	r

M/R — 0 for register to memory, 1 for memory to register
s — size, 0 word, 1 long
m — addressing mode
r — address register

this should be followed by an extra word defining which registers are to be moved. Each bit is set if the register is to be moved, else it is reset. The form of the word depends on the addressing mode, and is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

except for pre-decrement, when the word is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

Addressing modes:

Memory to register: MOVEM (address),(register)

Mode	mmm	rrr
(An)	010	reg no.
(An)+	011	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011

Register to Memory: MOVEM (registers),(address)

Mode	mmm	rrr
(An)	010	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: the primary use for this instruction is saving and restoring registers from the stack, using pre-decrement and post-increment addressing with A7. The opcodes are

$48E7 for MOVEM.L (registers),−(A7)
$4CDF for MOVEM.L (A7)+,(registers)

both being followed by a word defining the register list.

MOVEP — MOVE PERIPHERAL

Mnemonic: MOVEP Dx,(d)(Ay) and MOVEP (d)(Ay),Dx

Size: word or long

Action: this is for transferring data to and from memory, using only the even or odd numbered addresses. The highest order byte is transferred first. This is intended for use in other processors in the 68000 series, so that they can use 8 bit peripheral devices more easily.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	x	x	x	1	om		s	0	0	1	y	y	y

x — data register number
om — 0 for memory to register, 1 for register to memory
s — size, 0 word, 1 long
y — address register number

This should be followed by an extra word containing the displacement to be added to the address registers.

Notes: because the 68008 has an 8 bit data bus anyway, this instruction is redundant on the QL.

MOVEQ — MOVE QUICK

Mnemonic: MOVEQ #(d),Dx

Size: long only

Action: the immediate data is transferred into the data register. The range of values of the data is 0–$7F and $FFFFFFFF to $FFFFFF80 inclusive. It is faster and takes less bytes than its MOVE equivalent.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	x	x	x	0	d	d	d	d	d	d	d	d

x — data register number
d — 8 bit immediate data, sign extended to *32 bits*

MULS — SIGNED MULTIPLICATION

Mnemonic: MULS (address),Dx

Size: word only

Action: the word contents of the address is multiplied by the word contents of the register, and the 32 bit result put into the data register. The multiplication is done using signed arithmetic.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	x	x	x	1	1	1	m	m	m	r	r	r

x — data register number
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: all addressing modes are allowed, except address register direct. An overflow or a carry cannot occur because it is impossible to multiply two 16 bit numbers together and get a result greater than 32 bits. This is another instruction that takes a varying amount of time to execute, depending on the patterns of bits within the contents of the address.

MULU — UNSIGNED MULTIPLICATION

Mnemonic: MULU (address),Dx

Size: word only

Action: the word contents of the address are multiplied by the word contents of the data register, and the 32 bit result put into the data register. It is done using unsigned arithmetic.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	x	x	x	0	1	1	m	m	m	r	r	r

x — data register number
m — addressing modes
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: all addressing modes bar address register direct are allowed. The time taken for this to execute is proportional to the number of set bits in the contents of the address.

NBCD — NEGATE BINARY CODED DECIMAL WITH EXTEND

Mnemonic: NBCD (address)

Size: byte only

Action: the BCD pair in the address is ten's-complemented, and the state of the extend flag subtracted too, ie the calculation 0-address contents-X is done, in base 10.

Condition codes:

X	N	Z	V	C
*	?	*	?	*

Z — reset if the result is non-zero, else *unchanged*
C — set if a decimal carry was generated

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	m	m	m	r	r	r

m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

NEG — NEGATE

Mnemonic: NEG (address)

Size: byte, word or long

Action: the contents of the address are subtracted from 0, and the result stored in the address. In other words, the data is two's-complemented.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: Don't confuse this one with the NOT instruction.

NEGX — NEGATE WITH EXTEND

Mnemonic: NEGX (address)

Size: byte, word or long

Action: the contents of the address are subtracted from 0, and the state of the extend bit also subtracted, ie the operation 0-contents of address-X.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: this is the base 16 equivalent of NBCD.

NOP — NO OPERATION

Mnemonic: NOP

Size: n/a

Action: nothing is done. The next instruction is executed after a very short delay.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1	=$4E71

Notes: this is usually used in timing delays, and for removing instructions when debugging.

NOT — COMPLEMENT

Mnemonic: NOT (address)

Size: byte, word or long

Action: the contents of the address are complemented, and placed back in the address, ie the state of each bit is inverted.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

OR — LOGICAL OR

Mnemonic: OR (address),Dx and OR Dx,(address)

Size: byte, word or long

Action: the two parameters are logically ORed, and the result placed back in either the data register (former mnemonic) or the address (latter mnemonic).

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	x	x	x	om	s	s	m	m	m	r	r	r

x — data register number
om — 0 for OR (address),Dx
1 for OR Dx,(address)
s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no. (not Dx,(address))
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010 (not Dx,(address))
n(PC,A/Dn)	111	011 (not Dx,(address))
#nn	111	100 (not Dx,(address))

Notes: there are other forms of logical OR — ORI, for immediate data, and OR to CCR and SR.

ORI — LOGICAL OR IMMEDIATE

Mnemonic: ORI #(d),(address)

Size: byte, word or long

Action: the immediate data is logically ORed with the contents of the address, and the result placed back in the address. The size of the data matches the size of the instruction.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register
The opcode should be followed by one or two words containing the immediate data. For byte operations, the data should be in the lower byte of the extra word.

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

ORI TO SR/CCR — LOGICAL OR IMMEDIATE TO THE STATUS REGISTER

Mnemonic: ORI #(d),CCR and ORI #(d),SR

Size: byte (for CCR) or word (for SR)

Action: the immediate data is logically ORed with part or all of the status register, and the result placed back into the

same part of the SR. *ORI to the SR is a privileged instruction.*

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The state of the condition codes depends on the combination of the immediate data and its previous values.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	x	1	1	1	1	0	0

x — 0 for CCR (byte), 1 for SR (word)
followed by an extra word containing the immediate data (in the low byte if CCR).

PEA — PUSH EFFECTIVE ADDRESS

Mnemonic: PEA (address)

Size: long only

Action: the value of (address) is calculated (in the same way as LEA), then the long word pushed on to the stack. It is equivalent to the instructions

LEA (address),Ax
MOVEA.L Ax,−(A7)

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	m	m	m	r	r	r

m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
(An)	010	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011

Notes: This is a seldom used instruction.

RESET — RESET

Mnemonic: RESET

Size: n/a

Action: a RESET signal is sent from the processor to other external devices. This is a privileged instruction. The processor itself is *not* reset.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	=$4E70

Notes: because of the hardware within it, this instruction does in fact cause a system reset on the QL.

ROL/R — ROTATE LEFT AND RIGHT

Mnemonics: ROL Dx,Dy and ROL #(d),Dx and ROL (address) and
ROR Dx,Dy and ROR #(d),Dx and ROR (address)

Size: byte, word or long (word only for (address))

Action: a rotate is one in which the operand has a number of bits rotated to the left or right, with the end bit rotating around to the other end. This is best illustrated by a diagram (see **Figure 8.3**). The number of rotates and the item rotated can be specified in a number of ways: using Dx,Dy register Dy is rotated by the value of Dx, modulus 64. Using #(d),Dx register Dx is rotated a number of times depending on the immediate data, from 1 to 8. Using just (address), the contents of the

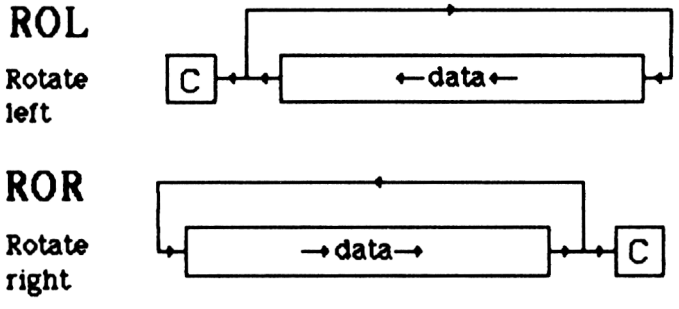


Figure 8.3: Rotates.

address are rotated once only, and the size is always word. Only the relevant part of the parameter is rotated, depending on the size.

Condition codes:

X	N	Z	V	C
–	*	*	0	*

N — set if the highest bit of the result is set
Z — set if the result is zero, else cleared
C — set according to **Figure 8.3**, or unaffected by a zero rotate count.

Opcodes:

for Dx,Dy:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	x	x	x	l/r	s	s	1	1	1	y	y	y

x — first register (count register) number
l/r — direction, 0 for right, 1 for left
s — size, 00 byte, 01 word, 10 long
y — second register number (the one rotated)

for #(d),Dx:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	d	d	d	l/r	s	s	0	1	1	x	x	x

d — immediate data, from 0–7, 0 gives a rotate count of 8
l/r — direction 0 for right, 1 for left
s — size, 00 byte, 01 word, 10 long
x — data register number

for (address) (ie memory rotates):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	l/r	1	1	m	m	m	r	r	r

l/r — direction, 0 for right, 1 for left
m — addressing mode
r — address register

Addressing modes: (for memory rotates only)

Mode	mmm	rrr
(An)	010	reg no.
(An)+	011	reg no.
–(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

ROXL/R — ROTATE LEFT AND RIGHT WITH EXTEND

Mnemonic: ROXL Dx,Dy and ROXL #(d),Dx and ROXL (address) and
ROXR Dx,Dy and ROXR #(d),Dx and ROXR (address)

Size: byte, word or long (word only for (address))

Action: this is similar to ROL and ROR, but the Extend flag is used instead of the Carry flag, as in **Figure 8.4**. The number of rotates and the item rotated can be specified in a number of ways. Using Dx,Dy, register Dy is rotated by the value of Dx, modulus 64. Using #(d),Dx, register Dx is rotated a number of times depending on the immediate data, from 1 to 8. Using just (address), the contents of the address are rotated once only, and the size is always word. Only the relevant part of the parameter is rotated, depending on the size.

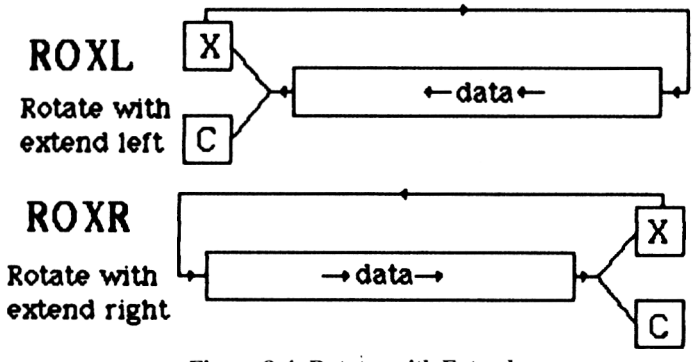


Figure 8.4: Rotates with Extend.

Condition codes:

X	N	Z	V	C
*	*	*	0	*

X — set according to the diagram, or unaffected by a zero rotate count
N — set if the highest bit of the result is set
Z — set if the result is zero, else cleared
C — set according to the diagram, or unaffected by a zero rotate count

Opcodes: *for Dx,Dy:*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	x	x	x	l/r	s	s	1	1	0	y	y	y

x — first register (count register) number
l/r — direction, 0 for right, 1 for left
s — size, 00 byte, 01 word, 10 long
y — second register number (the one rotated)

for #(d),Dx:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	d	d	d	l/r	s	s	0	1	0	x	x	x

d — immediate data, from 0–7, 0 gives a rotate count of 8
l/r — direction 0 for right, 1 for left
s — size, 00 byte, 01 word, 10 long
x — data register number

for (address) (ie memory rotates):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	l/r	1	1	m	m	m	r	r	r

l/r — direction, 0 for right, 1 for left
m — addressing mode
r — address register

Addressing modes (for memory rotates only):

Mode	mmm	rrr
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

RTE — RETURN FROM EXCEPTION

Mnemonic: RTE

Size: n/a

Action: the top word on the stack is removed and put in the status register, then a long word pulled from the stack and execution commences at that address. This is for returning from exception-handling routines, and *is a privileged instruction.*

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The state of the condition codes depends on the top word pulled from the stack.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1	=$4E73

Notes: this instruction does not see which exception caused it, so for address errors and bus errors it is up to the programmer to remove the four extra words placed on the stack before doing the RTE.

RTR — RETURN AND RESTORE

Mnemonic: RTR

Size: n/a

Action: the word on the stack is removed, and the lower byte put in the status register. Then a long word is removed from the stack, and execution commences at the address. As the high byte of the SR is not altered, this is not a privileged instruction.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The condition codes depend on the word removed from the stack.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1	=$4E77

Notes: this instruction is normally used for leaving subroutines that originally saved the state of the SR on the stack with MOVE SR,–(A7).

RTS — RETURN FROM SUBROUTINE

Mnemonic: RTS

Size: n/a

Action: the long word on the stack is pulled, and a jump made to it.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1	=$4E75

Notes: a very common instruction indeed, especially for returning to BASIC after a CALL command.

SBCD — SUBTRACT BINARY CODED DECIMAL

Mnemonic: SBCD Dx,Dy and SBCD −(Ax),−(Ay)

Size: byte only

Action: the first BCD parameter is subtracted from the second in base 10, then the state of the extend flag subtracted, and the result placed in the second. The parameters can be expressed either as data registers, or by address register pre-decrement addressing.

Condition codes:

X	N	Z	V	C
*	?	*	?	*

Z — zeroed if the result is zero, else *unchanged*
C — set if a decimal carry was generated
X — same as C

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	y	y	y	1	0	0	0	0	R/M	x	x	x

y — first data register number
R/M — 0 for Dx,Dy, 1 for −(Ax),−(Ay)
x — second register number

SCC — SET IF CONDITION

Mnemonic: Scc (address)
where 'cc' is a condition

Size: byte only

Action: if the specified condition is true, then the contents of the address will be set to $FF, else it will be set to $00.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	c	c	c	c	1	1	m	m	m	r	r	r

c — condition, one of:

0000	T
0001	F
0010	HI
0011	LS
0100	CC
0101	CS
0110	NE
0111	EQ

1000 VC
1001 VS
1010 PL
1011 MI
1100 GE
1101 LT
1110 GT
1111 LE

m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: the instruction SF (address) will always zero the address, and ST (address) will always set it to $FF. When accessing memory, the memory is always read before being written to.

STOP — STOP

Mnemonic: STOP#(d)

Size: word only

Action: the immediate data is put into the status register, the program counter is incremented to point to the next instruction, and the processor literally stops. As soon as any exception is required, the processor will resume — the normal exception to cause this is an interrupt, but if the trace bit is set prior to the STOP, the processor will re-start. The other way of re-starting a stopped processor is by a signal applied to the RESET pin, which will cause a complete processor reset. *This instruction is privileged.*

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The state of the condition codes depends on the immediate data in the instruction.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0	=$4E72

followed by one extra word, containing the immediate data.

Notes: on the QL, this instruction will permanently halt the processor unless either the tracing is enabled, or the interrupt mask is less than 3.

SUB — SUBTRACT

Mnemonic: SUB (address),Dx and SUB Dx,(address)

Size: byte, word or long

Action: the first parameter is subtracted from the second, and the result placed back in the second. Each parameter may be either a data register, or an addressing mode.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	x	x	x	om	s	s	m	m	m	r	r	r

x — data register number
om — 0 for SUB (address),Dx
1 for SUB Dx,(address)
s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no. (not Dx,(address), use (address),Dx)
An	001	reg no. (not Dx,(address) (use SUBA) and not byte)
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010 (not Dx,(address))
n(PC,A/Dn)	111	011 (not Dx,(address))
#nn	111	100 (not Dx,(address))

Notes: there are four other types of SUB — SUBA, for address registers, SUBI, for immediate data, SUBQ, for quick subtractions, and SUBX, for extended subtraction.

SUBA — SUBTRACT ADDRESS

Mnemonic: SUBA (address),Ax

Size: word or long

Action: the contents of the address are subtracted from the given address register, and the result put back in the address register. For word sizes, both the address and address registers' contents are sign-extended to 32 bits before the subtraction, giving a 32 bit result.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	x	x	x	s	1	1	m	m	m	r	r	r

x — address register number
s — size, 0 word, 1 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
An	001	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001
n(PC)	111	010
n(PC,A/Dn)	111	011
#nn	111	100

Notes: all addressing modes are allowed. None of the condition codes are affected by this instruction, unlike all other subtract instructions.

SUBI — SUBTRACT IMMEDIATE

Mnemonic: SUBI #(d),(address)

Size: byte, word or long

Action: the immediate data is subtracted from the contents of the address. The size of the data equals the size of the instruction.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

This has to be followed by one or two extra words, containing the immediate data. For byte-sized operations, the lower byte of the extra word contains the data.

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: address register direct is not allowed, you can use SUBA instead.

SUBQ — SUBTRACT QUICK

Mnemonic: SUBQ #(d),(address)

Size: byte, word or long

Action: the immediate data is subtracted from the contents of the address. The data can have a range of 1 to 8 inclusive. It is much faster and more memory efficient than the equivalent SUBI instruction. For word sized SUBA, both parameters are sign extended to 32 bits before the operation.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

There is no effect on the condition codes for SUBQ #(d),Ax.

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	x	x	x	1	s	s	m	m	m	r	r	r

x — data, a value of 0 gives a SUBQ of 8
s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
An	001	reg no. (not byte)
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

SUBX — SUBTRACT WITH EXTEND

Mnemonic: SUBX Dx,Dy and SUBX −(Ax),−(Ay)

Size: byte, word or long

Action: the first parameter is subtracted from the second, then the state of the extend flag subtracted, and the result stored in the second. Only two forms of parameters are allowed — data register direct, and address register indirect with pre-decrement.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	y	y	y	1	s	s	0	0	R/M	x	x	x

y — second register number
s — size, 00 byte, 01 word, 10 long
R/M — 0 for Dx,Dy, 1 for −(Ax),−(Ay)
x — first register number

SWAP — SWAP HIGH AND LOW WORDS

Mnemonic: SWAP Dx

Size: n/a

Action: the low word of the data register and the high word are swapped over.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Z — set if the whole register is 0
N — set if bit 31 of the result is set

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	x	x	x

x — data register number

Notes: this can be very useful at times, especially if the register holds two word quantities, which can be swapped around for different word-sized operations, as the high word of registers is not affected by them.

TAS — TEST AND SET

Mnemonic: TAS (address)

Size: byte only

Action: the byte contents of the address are read, the condition codes set accordingly, and then bit 7 of the address contents is set.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	m	m	m	r	r	r

m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
−(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: this instruction is carried out by the hardware in an indivisible way, and is intended for use in multi-processor applications, for setting semaphores to handshake between processors. It is, however, not required on the QL for communicating with the 8049 IPC.

TRAP — TRAP

Mnemonic: TRAP#(number)

Size: n/a

Action: the trap instruction includes a parameter from 0 to 15 inclusive, and causes an exception when executed. The vector taken for the exception depends on the number of the trap, namely 32+n, where n is the trap number.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	x	x	x	x

x — trap number, 0 to 15

Notes: trap-handling on the QL is covered in detail in Chapter 6, but to recap traps 0 to 4 are QDOS calls, and 5 to 15 can be defined by the programmer.

TRAPV — TRAP IF OVERFLOW

Mnemonic: TRAPV

Size: n/a

Action: if the overflow (V) flag is set, an exception will occur, using vector number 7.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0	=$4E76

Notes: on the QL, the exception is usually ignored, but can be user-defined.

TST — TEST

Mnemonic: TST (address)

Size: byte, word or long

Action: the contents of the address are compared to 0, and some of the condition codes altered.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

N — set if the parameter is negative
Z — set if the parameter is zero

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	s	s	m	m	m	r	r	r

s — size, 00 byte, 01 word, 10 long
m — addressing mode
r — address register

Addressing modes:

Mode	mmm	rrr
Dn	000	reg no.
(An)	010	reg no.
(An)+	011	reg no.
–(An)	100	reg no.
n(An)	101	reg no.
n(An,A/Dn)	110	reg no.
nn.W	111	000
nn.L	111	001

Notes: with QDOS system calls, it is usual for D0 to contain a negative number if an error has occurred, else it is 0 if there were no errors. The easiest (and fastest) way of testing it is by

TST.B D0

and then you can BEQ if it was OK, or BNE if it was not.

UNLK — UNLINK

Mnemonic: UNLK Ax

Size: n/a

Action: this is the opposite of the LINK instruction. A7 is loaded with the address register, and the address register loaded with the next long word pulled from the stack.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	x	x	x

x — address register number

Notes: as with LINK, this is an advanced command and will not be discussed further here.

CHAPTER 9

The 68008 Disassembler

When I began hand coding on the QL, I found I needed a reliable way of checking that I had made no mistakes. I also needed to inspect the ROM, particularly the exception-handling routines, and the obvious solution to both problems was to write a disassembler.

A disassembler is the opposite of an assembler — it converts a section of memory from numbers back into recognisable instructions. For speed and ease of debugging, it was written initially in SuperBASIC, but I have since converted it to machine code. In this chapter I shall explain the method, give the full BASIC listing, and give guidelines for converting it into 68008 instructions.

Disassemblers for older processors tend to be easy things to write — there are usually a fixed number of instructions, such as 256 or 512, and a big data table along with a few extra lines is usually sufficient. However, on the 68008 life is rather harder, as there are many thousands of different instructions, because of the different addressing modes. Instead of using a huge data table, a small one can be used, and then lines added to work out the addressing modes used, and convert them to instructions.

The algorithm

The most important part of the algorithm is the 'search for the opcode type' table. As can be seen from Chapter 8, each instruction is based on a 16 bit word, with certain bits fixed, and others variable. The fixed bits determine the basic instruction, while the variable ones are usually things like addressing modes and data. There are bound to be other ways of doing this, but I used a table of about 70 different entries, each consisting of four items — the fixed bits of the opcode, a 'mask' showing which of the bits are fixed, a string giving the first part of the mnemonic, and finally a number from 0 to 31 giving the 'type'. For example, the instruction JMP has the following opcode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	m	m	m	r	r	r

Bits 6 through 15 are fixed, while bits 0 to 5 vary, according to the addressing mode used. This is represented in the table by the entry

8410 DATA 20160, 65472, "JMP .",12

It is a little clearer if you convert the numbers into hex and binary thus:

20160	$4EC0	0100	1110	1100	0000
65472	$FFC0	1111	1111	1100	0000

The first word contains the fixed bit, with the variable bits set to 0. The second word has its bits set if the corresponding bit is fixed in the opcode, else the bit is reset. The string 'JMP' is the start of the instruction (with a full stop added for reasons to be explained later), and 12 denotes the 'type'. My investigations initially made me think there were 32 different types of instruction. By type I mean that different mnemonics decode the same way, except for their start — for example, another type 12 instruction is JSR, which you can see from its opcode format. After some time, I found I could eliminate some types further, so that now there are around 28 at the last count. A bit more work could probably reduce it further, but I did not feel it was necessary.

Some of the BASIC procedures and functions are not as efficient as they could be. This is because the program was written and debugged on an early version of the QL (version FB to be exact), which suffered from a very bug-ridden BASIC. It was particularly sensitive to local variables and FOR loops, which accounts for some of the quirks you may find in the listing. As it was written on the first QL variant, it has the advantage of working on all the later models (I hope!). Another problem was that I did not originally realise that the functions PEEK_W and PEEK_L gave signed results, so the procedure POS was written to ensure that certain operations gave positive results.

Now, the full listing. As with all programs, you should type it in in stages, saving it as you go along in case of system failure, power cuts, someone unplugging you to plug in their hairdryer, etc. I made it as 'structured' as I could, though a couple of the dreaded GOTOs still remain! When I remembered I used leading spaces to indent lines for clarity.

Listing 9.1: The disassembler

```
1 CLS:PRINT "          DISSASSEMBLER"\\
2 PRINT "    (c) Andrew Pennell 1984"
5 nop=72:DIM oplist(nop,3),op$(nop,12)
```

```
6 RESTORE 8000
7 FOR i=1 TO nop:READ oplist(i,1),oplist
(i,2):READ op$(i):READ oplist(i,3)
10 INPUT "Start address ($ for hex) ";a$
:IF a$="" THEN GO TO 10
11 INPUT "Output stream (normally 1) ";s
trm:PRINT #strm
15 IF a$(1)="$" THEN dec(a$(2 TO )):pc=d
d: ELSE pc=a$
20 diss(pc)
30 IF INKEY$=" " THEN GO TO 10:ELSE GO T
O 20
100 DEFine FuNction h1$(a)
110  RETurn CHR$(48+a+7*(a>9))
120 END DEFine
130 DEFine FuNction h2$(a)
150  RETurn h1$(a DIV 16)&h1$(a MOD 16)
160 END DEFine
170 DEFine FuNction hex$(b)
175  LOCal a,h$
177  a=pos(b):IF a>32767 THEN a=a-32768
180  h$=h2$(a DIV 256)&h2$(a MOD 256)
185  IF pos(b)>32767 THEN h$(1)=h1$(h$(1
)+8)
187  RETurn h$
190 END DEFine
200 DEFine FuNction band(x,y)
210  LOCal a,b,c,d
220  a=INT(pos(x)/32768):b=INT(pos(y)/32
768)
230  c=sml(pos(x)):d=sml(pos(y))
240  RETurn 32768*(a&&b)+(c&&d)
250 END DEFine
260 DEFine FuNction sml(z)
270  RETurn z-32768*INT(z/32768)
280 END DEFine
300 DEFine PROCedure dec(a$)
310  LOCal s,t,q
320  t=0
330  q=LEN(a$)
340  s=CODE(a$(q))-48:IF s>22 THEN s=s-3
2
345  IF s>9 THEN s=s-7
350  t=t+s*16^(LEN(a$)-q)
```

```
360  q=q-1:IF q>0 THEN GO TO 340
365  dd=t
370  RETurn
380 END DEFine
400 DEFine FuNction BHEX$(a)
410  LOCal h1,h2
420  h1=INT(a/65536):h2=a-65536*h1
430  RETurn hex$(h1)&hex$(h2)
440 END DEFine
450 DEFine FuNction pos(a)
460  IF a<0 THEN RETurn 65536+a:ELSE RET
urn a
470 END DEFine

500 DEFine PROCedure reglist(x,t$)
505  LOCal bit
510  IF x=0 THEN RETurn
520  bit=7
530  IF NOT(x&&(2^bit)) THEN GO TO 650
540  p$=p$&t$&(7-bit)
550  IF bit=0 THEN p$=p$&"/":GO TO 650
560  bit=bit-1:IF NOT(x&&(2^bit)) THEN p
$=p$&",":GO TO 650
570  bit=bit-1
580  IF NOT(x&&(2^bit)) THEN p$=p$&"-"&t
$&(6-bit)&"/":GO TO 650
590  IF bit=0 THEN p$=p$&"-"&t$&"7/":GO
TO 650
600  GO TO 570
650  IF bit=0 THEN p$=p$(1 TO LEN(p$)-1)
:RETurn
660  bit=bit-1
665  GO TO 530
670 END DEFine
680 DEFine FuNction back(a)
690  LOCal b
700  b=0
710  FOR i=0 TO 7:IF a&&(2^(7-i)) THEN b
=b+2^i
720  RETurn b
730 END DEFine
1000 DEFine PROCedure diss(start)
1010  pc=start:op=PEEK_W(pc):pc=pc+2:mem
=start:IF op<0 THEN op=op+65536
```

```
1015  PRINT #strm;BHEX$(start);" ";
1022  i=0
1030  REPeat getlp:i=i+1:IF band(op,opli
st(i,2))=oplist(i,1) THEN EXIT getlp
1040  p$=op$(i):type=oplist(i,3)
1060  do_op(type)
1090  FOR i=mem TO pc-1
1100  PRINT #strm;h2$(PEEK(i));
1110  NEXT i
1115  FOR i=pc-mem TO 6:PRINT #strm;"   "
;
1120  PRINT #strm;p$
1130 END DEFine
1999 REMark eff addr calc
2000 DEFine PROCedure ea(amode)
2001 REMark reg must a var
2010 SELect ON amode
2020 =0
2030  p$=p$&"D"&reg
2040 =1
2050  p$=p$&"A"&reg
2060 =2
2070  p$=p$&"(A"&reg&")"
2080 =3
2090  p$=p$&"(A"&reg&")+"
2100 =4
2110  p$=p$&"-(A"&reg&")"
2120 =5
2140  p$=p$&h2$(PEEK(pc))&h2$(PEEK(pc+1)
)&"(A"&reg&")"
2145  pc=pc+2
2150 =6
2160  byte=PEEK_W(pc):pc=pc+2
2170  d=band(byte,255)
2180  p$=p$&h2$(d):p$=p$&"(A"&reg&","
2190  IF byte<0 THEN
2200   p$=p$&"A"
2210  ELSE
2220   p$=p$&"D"
2230  END IF
2240  reg=INT(band(byte,7*4096)/(2^12))
2250  d=band(byte,2^11)
2260  IF d THEN
2270   p$=p$&reg&".L)"
```

```
2280  ELSE
2290   p$=p$&reg&".W)"
2300  END IF
2320 =7
2330  REMark MODE 7S HERE
2340  SELect ON reg
2350   ON reg=0
2360    p$=p$&hex$(PEEK_W(pc))
2370    pc=pc+2
2380   ON reg=1
2390    p$=p$&BHEX$(PEEK_L(pc))
2400    pc=pc+4
2410   ON reg=2
2420    byte=PEEK_W(pc)
2430    byte=byte+pc:pc=pc+2
2440    p$=p$&BHEX$(byte)&"(PC)"
2450   ON reg=3
2460    byte=PEEK_W(pc)
2470    p$=p$&BHEX$(pc+comp2(PEEK(pc+1))
)&"(PC,"
2480    pc=pc+2
2490    IF band(byte,2^15) THEN p$=p$&"A
":ELSE p$=p$&"D"
2500    p$=p$&(band(byte,7*4096)/4096)&"
."
2510    IF band(byte,2^11) THEN p$=p$&"L
)":ELSE p$=p$&"W)"
2600   ON reg=4
2620    p$=p$&"#"
2630    IF size=1 THEN p$=p$&h2$(PEEK(pc
+1)):pc=pc+2
2640    IF size=2 THEN p$=p$&BHEX$(PEEK_
L(pc)):pc=pc+4
2650    IF size=3 THEN p$=p$&hex$(PEEK_W
(pc)):pc=pc+2
2655   ON reg=REMAINDER
2656    p$=p$&"?M7,R"&reg&"?"
2660   END SELect
3900 =REMAINDER
3910  p$=p$&"????"
3920 END SELect
3930 END DEFine ea
3940 DEFine PROCedure efad(op)
3950 reg=band(op,7):m=band(op,7*8)/8
```

```
3960 ea(m)
3970 END DEFine efad
4000 DEFine PROCedure do_op(type)
4010 SELect ON type
4020 ON type=0
4030 REMark **** 0 ****SIMPLE***
4200 ON type=2
4210 REMark **** 2 **OR,AND,SUB,ADD,II**
*
4220  size=band(op,192)/64:do_dot(size)
4230  p$=p$&"#"
4240   SELect ON size
4250   ON size=0:p$=p$&h2$(PEEK(pc+1)):p
c=pc+2
4260   ON size=1,2:p$=p$&hex$(PEEK_W(pc)
):pc=pc+2
4270   END SELect
4280  IF size=2 THEN p$=p$&hex$(PEEK_W(p
c)):pc=pc+2
4290  p$=p$&","
4300  IF band(op,63)=60 THEN
4304   IF size=0 THEN p$=p$&"CCR":ELSE p
$=p$&"SR"
4306  ELSE
4308   efad(op)
4310  END IF
4315 ON type=3
4325 REMark **** 3 ****  MOVEP
4335  IF band(op,2^6) THEN p$=p$&"L ":EL
SE p$=p$&"W "
4340  a$="D"&(band(op,7*2^9)/(2^9)):B$=h
ex$(PEEK_W(pc))&"(A"&band(op,7)&")"
4345  pc=pc+2:IF band(op,2^7) THEN p$=p$
&a$&","&B$:ELSE p$=p$&B$&","&a$
4350 ON type=4
4355 REMark **** 4 **** Bxxx,Dn,ea
4360  bits(band(op,192)/64):size=0
4365  p$=p$&"D"&(band(op,7*2^9)/(2^9))&"
,"
4370  efad(op)
4375 ON type=5
4380 REMark **** 5 **** Bxx,#d,ea
4385  bits(band(op,192)/64):size=0
4390  p$=p$&"#"&h2$(PEEK(pc+1))&","
```

```
4395  pc=pc+2:efad(op)
4500 ON type=6
4510 REMark **** 6 **MOVE
4535  size=band(op,2^13+2^12)/(2^12)
4540  efad(op):REMark source
4550  p$=p$&","
4560  reg=band(op,7*2^9)/(2^9)
4570  m=band(op,448)/(2^6)
4580  ea(m):REMark destination
4600 ON type=7
4610 REMark **** 7 **** STOP
4620  p$=p$&hex$(PEEK_W(pc))
4630  pc=pc+2
4800 ON type=8
4810 REMark **** 8 **** SWAP,EXT,UNLK
4820  p$=p$&band(op,7)
4830 ON type=9
4840 REMark **** 9 **** LINK
4850  p$=p$&band(op,7)&",#"
4860  p$=p$&hex$(PEEK_W(pc))
4870  pc=pc+2
5000 ON type=10
5010 REMark **** 10 **** MOVEAn,USP
5020  p$=p$&band(op,7)&",USP"
6100 ON type=11
6110 REMark **** 11 **** TRAP
6120  p$=p$&band(op,15)
6200 REMark
6210 ON type=12
6220 REMark **** 12 **** TST,JSR,JMP
6230  IF p$(LEN(p$))<>"," THEN size=band
(op,192)/64:do_dot(size): ELSE size=2:p$
=p$(1 TO LEN(p$)-1)
6250  efad(op)
6255 ON type=13
6260 REMark **** 13 **** MOVE to CCR
6265  size=1:efad(op)
6270  p$=p$&",CCR"
6275 ON type=14
6280 REMark **** 14 **** MOVE to SR
6285  size=1:efad(op)
6290  p$=p$&",SR"
6300 ON type=15
6310 REMark **** 15 **** MOVEM
```

```
6320  size=1+band(op,2^6)/(2^6):do_dot(s
ize)
6330  d1=back(PEEK(pc)):d2=back(PEEK(pc+
1)):pc=pc+2:efad(op):p$=p$&","
6340  reglist d1,"A":IF d1<>0 THEN p$=p$
&"/"
6350  reglist d2,"D"
6405 ON type=16
6410 REMark **** 16 **** MOVEM#2
6415  size=1+band(op,2^6)/(2^6):do_dot(s
ize)
6420  d1=PEEK(pc):d2=PEEK(pc+1):pc=pc+2
6430  reglist d1,"D":IF d1<>0 THEN p$=p$
&"/"
6440  reglist d2,"A"
6450  p$=p$&",":efad(op)
6455 ON type=17
6460 REMark **** 17 **** MUL,CHK,DIV
6465  size=2:efad(op)
6470  reg=band(op,7*2^9)/(2^9)
6475  p$=p$&",D"&reg
6500 ON type=18
6510 REMark **** 18 **** LEA
6520  size=2
6530  efad(op)
6540  reg=band(op,7*2^9)/(2^9)
6550  p$=p$&",A"&reg
6555 ON type=19
6560 REMark **** 19 **** DBcc
6565  do_cond(band(op,7*2^8)/(2^8))
6570  p$=p$&"D"&band(op,7)&","
6575  p$=p$&BHEX$(pc+comp2_w(PEEK_W(pc))
)
6580  pc=pc+2
6600 ON type=20
6610 REMark **** 20 **** Scc
6620  CON=band(op,7*2^8)/(2^8):do_cond(C
ON)
6630  efad(op)
6700 ON type=21
6710 REMark **** 21 **** ADDQ,SUBQ
6720  size=band(op,192)/64:do_dot(size)
6730  d=band(op,7*2^9)/(2^9):d=d+8*(d=0)
:p$=p$&"#"&d&","
```

```
6750  efad(op)
6900 ON type=23
6910 REMark **** 23 **** BRA,Bcc,BSR
6920  CON=band(op,15*2^8)/(2^8)
6930  SELect ON CON
6940   ON CON=0:p$="BRA "
6950   ON CON=1:p$="BSR "
6960   ON CON=2 TO 15:do_cond(CON)
6970  END SELect
6980  byte=band(op,255)
6990  IF byte THEN
7000   p$=p$&BHEX$(pc+comp2(byte))
7010  ELSE
7020   p$=p$&BHEX$(pc+comp2_w(PEEK_W(pc)
))
7030   pc=pc+2
7040  END IF
7050 ON type=24
7060 REMark **** 24 **** MOVEQ
7070  p$=p$&h2$(band(op,255))&",D"
7080  d=band(op,7*2^9)/2^9:p$=p$&d
7083 ON type=25
7086 REMark **** 25 **** Dx,Dy/-(Ay)
7088  IF p$(LEN(p$))="X" THEN do_dot(ban
d(op,192)/64):ELSE p$=p$&" "
7090  rx=band(op,7*2^9)/(2^9):ry=band(op
,7)
7092  IF band(op,2^3) THEN
7094   p$=p$&"-(A"&ry&"),-(A"&rx&")"
7096  ELSE
7098   p$=p$&"D"&ry&",D"&rx
7099  END IF
7100 ON type=28
7110 REMark **** 28 **** CMPM
7120  size=band(op,3*2^6)/(2^6):do_dot(s
ize)
7130  reg=band(op,7):p$=p$&"(A"&reg&")+,
(A"
7140  reg=band(op,7*2^9)/(2^9):p$=p$&reg
&")+"
7200 ON type=29
7210 REMark **** 29 **** CMP.CMPA,EOR
7220  om=band(op,7*64)/64:areg=band(op,7
*2^9)/(2^9)
```

```
7230  SELect ON om
7240   ON om=3,7
7250    size=1+(om=7):p$="CMPA":do_dot(s
ize)
7260    efad(op):p$=p$&",A"&areg
7270   ON om=0 TO 2
7280    size=om:p$="CMP":do_dot(size)
7290    efad(op):p$=p$&",D"&areg
7300   ON om=4 TO 6
7310    size=om&&3:p$="EOR":do_dot(size)
7320    p$=p$&"D"&areg&",":efad(op)
7330   END SELect
7340 ON type=22
7350 REMark **** 22 **** EXG
7360  om=band(op,248)/8:rx=band(op,7*2^9
)/(2^9)
7370  ry=band(op,7)
7380   SELect ON om
7390    ON om=8:p$=p$&"D"&rx&",D"&ry
7400    ON om=9:p$=p$&"A"&rx&",A"&ry
7410    ON om=17:p$=p$&"D"&rx&",A"&ry
7420    ON om=REMAINDER :p$=p$&"??,??"
7430   END SELect
7500 ON type=30,31
7510 REMark **** 30,31 **** ADD,SUB ****
7520  size=band(op,192)/64
7540  areg=band(op,7*2^9)/(2^9)
7545  IF size<>3 THEN
7546   REMark *ADD*
7547   do_dot(size)
7550   d=band(op,2^8)
7560    IF NOT d THEN
7570     REMark ea,Dn
7580     efad(op)
7590     p$=p$&",D"&areg
7600    ELSE
7610   p$=p$&"D"&areg&","
7620   efad(op)
7630    END IF
7640  ELSE
7650   REMark *ADDA*
7662   size=1+NOT(NOT(band(op,2^8))):do_
dot(size)
7665   efad(op)
```

```
7670    p$=p$&",A"&areg
7680   END IF
7700 ON type=32
7710 REMark **** 32 **** shifts&rots
7720   rot=band(op,24)/8
7730    SELect ON rot
7740     ON rot=0:p$="AS"
7750     ON rot=1:p$="LS"
7760     ON rot=2:p$="ROX"
7770     ON rot=3:p$="RO"
7780    END SELect
7790   IF band(op,2^8) THEN p$=p$&"L" :EL
SE p$=p$&"R"
7800   size=band(op,192)/64
7810   IF size<>3 THEN
7820    REMark reg rot
7830    do_dot(size):d=band(op,7*2^9)/2^9
:IF band(op,2^5) THEN p$=p$&"D"&d:ELSE p
$=p$&"#"&(d+8*(d=0))
7840    p$=p$&",D"&band(op,7)
7850   ELSE
7860    REMark memory rot
7870    efad(op)
7880   END IF
7900 =REMAINDER
7910   p$="dont know yet"
7920 END SELect
7930 RETurn
7940 END DEFine
8060 DATA 264,61752,"MOVEP.",3
8070 DATA 0,65280,"ORI",2
8080 DATA 512,65280,"ANDI",2
8090 DATA 1024,65280,"SUBI",2
8100 DATA 1536,65280,"ADDI",2
8110 DATA 2048,65280,"B",5
8120 DATA 2560,65280,"EORI",2
8130 DATA 3072,65280,"CMPI",2
8140 DATA 256,61696,"B",4
8150 DATA 4096,61440,"MOVE.B ",6
8160 DATA 8256,61888,"MOVEA.L ",6
8170 DATA 8192,61440,"MOVE.L ",6
8180 DATA 12352,61888,"MOVEA.W ",6
8190 DATA 12288,61440,"MOVE.W ",6
8200 DATA 19196,65535,"ILLEGAL",0
```

```
8210 DATA 20080,65535,"RESET",0
8220 DATA 20081,65535,"NOP",0
8230 DATA 20082,65535,"STOP #",7
8240 DATA 20083,65535,"RTE",0
8250 DATA 20085,65535,"RTS",0
8260 DATA 20086,65535,"TRAPV",0
8270 DATA 20087,65535,"RTR",0
8280 DATA 18496,65528,"SWAP D",8
8290 DATA 18560,65528,"EXT.W D",8
8300 DATA 18624,65528,"EXT.L D",8
8310 DATA 20048,65528,"LINK A",9
8320 DATA 20056,65528,"UNLK A",8
8330 DATA 20064,65528,"MOVE.L A",10
8340 DATA 20072,65528,"MOVE.L USP,A",8
8345 DATA 20032,65520,"TRAP #",11
8350 DATA 16576,65472,"MOVE.W SR,",12
8360 DATA 17600,65472,"MOVE.B ",13
8370 DATA 18112,65472,"MOVE.W ",14
8380 DATA 18432,65472,"NBCD .",12
8390 DATA 19136,65472,"TAS .",12
8400 DATA 20096,65472,"JSR .",12
8410 DATA 20160,65472,"JMP .",12
8420 DATA 18560,65408,"MOVEM",16
8430 DATA 19584,65408,"MOVEM",15
8440 DATA 18496,65344,"PEA .",12
8450 DATA 16384,65280,"NEGX",12
8460 DATA 16896,65280,"CLR",12
8470 DATA 17408,65280,"NEG",12
8480 DATA 17920,65280,"NOT",12
8490 DATA 18944,65280,"TST",12
8500 DATA 16768,61888,"CHK ",17
8510 DATA 16832,61888,"LEA ",18
8520 DATA 20680,61688,"DB",19
8530 DATA 20672,61632,"S",20
8540 DATA 20480,61696,"ADDQ",21
8550 DATA 20736,61696,"SUBQ",21
8580 DATA 24576,61440,"B",23
8590 DATA 28672,61696,"MOVEQ #",24
8600 DATA 33024,61936,"SBCD ",25
8605 DATA 32960,61888,"DIVU ",17
8610 DATA 33216,61888,"DIVS ",17
8630 DATA 32768,61440,"OR",31
8635 DATA 37056,61632,"SUBA",30
8640 DATA 37120,61744,"SUBX",25
```

```
8650 DATA 36864,61440,"SUB",31
8660 DATA 45320,64568,"CMPM",28
8670 DATA 45056,61440,"",29
8680 DATA 49472,61744,"EXG ",22
8710 DATA 49048,61936,"ABCD",25
8720 DATA 49344,61888,"MULU ",17
8730 DATA 49600,61888,"MULS ",17
8740 DATA 49152,61440,"AND",31
8745 DATA 53440,61632,"ADDA",30
8750 DATA 53504,61744,"ADDX",25
8760 DATA 53248,61440,"ADD",31
8770 DATA 57344,61440,"",32
8998 DATA 0,0,"??",0
8999 DATA -1,-1,"",-1
9000 DEFine PROCedure do_cond(N)
9010  RESTORE 9020
9020  DATA "T","F","HI","LS","CC","CS","
NE","EQ","VC","VS","PL","MI","GE","LT","
GT","LE"
9030  FOR i=0 TO N
9040   READ 0$
9050  NEXT i
9060  p$=p$&0$&" "
9070 END DEFine
9100 DEFine FuNction comp2(N)
9110  RETurn N-256*(N>127)
9120 END DEFine
9130 DEFine FuNction comp2_w(N)
9140  RETurn N-65536*(N>32768)
9150 END DEFine
9160 DEFine PROCedure do_dot(s)
9170  SELect ON s
9180   ON s=0:p$=p$&".B "
9190   ON s=1:p$=p$&".W "
9200   ON s=2:p$=p$&".L "
9220  END SELect
9230 END DEFine
9240 DEFine PROCedure bits(a)
9250  SELect ON a
9260   ON a=0:p$=p$&"TST"
9270   ON a=1:p$=p$&"CHG"
9280   ON a=2:p$=p$&"CLR"
9290   ON a=3:p$=p$&"SET"
9300  END SELect
```

```
9305  p$=p$&" "
9310 END DEFine
```

Before going through the listing, I'll explain the main variables. They are:

pc	the location of the next word to be disassembled
p$	the disassembled instruction
op	the first word of the instruction, ie the opcode
mem	the location of op
strm	the output stream (usually 1)
oplist( )	element 1 — opcode
	element 2 — fixed bit mask
	element 3 — the type
op$( )	the start of the instruction
nop	number of opcodes
type	the current instructions type
size	the current instructions size — 0 byte, 1 word, 2 long

Lines	Proc/Fn	Description
1–7		Read the data at lines 8000 ff into the arrays.
10–15		Choose start address and output stream.
20–30		The main loop disassembling instructions until space is pressed.
100–120	h1$	A sub-function for hex conversion.
130–160	h2$	Converts 0–255 into hex.
170–190	hex$	Converts 0–65535 into hex.
200–250	band	Does bitwise AND instruction on words — used instead of && as the latter doesn't work on numbers above $8000.
260–280	sml	Strips bit 15 off word (used by 'band').
300–370	dec	Converts hex string into decimal in 'dd'. (Note this is a PROC not an FN due to version FB problems.)
400–440	bhex$	Converts 0–$FFFFFFFF into hex.
500–670	reglist	Procedure to build up register list for MOVEM.
680–730	back	Function to reverse the bits in a byte.
1000–1130	diss	Main procedure. Starts by reading op, setting mem, and then prints start address. GETLP goes through the opcodes until the correct one is found, then p$ and type set. The full mnemonic

		is built up using do_op, then each opcode is printed.
2000–3930	ea	Important effective address calculator. Uses two important parameters — amode & reg, which represent the addressing mode and register number taken from the opcode byte. The SELECT command is used to distinguish between each mode, and later on between each register for mode 7.
2655		Prints strange mnemonic if an invalid register is specified in mode 7.
3930–3970	efad	Used for most instructions that have the register number in bits 0–2, and the addressing mode in bits 3 to 5. In fact the only instruction that doesn't is the destination part of the MOVE instruction.
4000–7940	do_op	Main code that calculates the remaining parts of each type of instruction, using SELect to distinguish between them.
4020		Type 0 instructions require no further additions, so no action is taken.
4200		Type 2 instructions namely the immediate forms OR, AND, SUB, ADD, including the CCR and SR forms.
4325		There is only one type 3 instruction — MOVEP.
4355		Type 4 instructions are the 'test a bit and ?' ones with dynamic bit numbers.
4380		Type 5 instructions are the 'test bit' ops with static bit numbers.
4510		Type 6 instructions are the MOVE family. For the source address, proc efad is used, but the destination uses proc ea.
4610		There is one type 7 instruction — STOP which requires a 16 bit parameter.
4810		Type 8 instructions require a following 3 bit parameter.
4830		The only type 9 instruction is LINK, and requires immediate data and register number.
5010		The only type 10 instruction is the MOVE to USP operation.

6110 TRAP is the only type 11 instruction, requiring a 4 bit parameter.

6220 Type 12 instructions are TST, JMP and JSR. TST is different as the size is variable, which is the reason for the full stops on the ends of JMP and JSR in the data table.

6260 Type 13 is the MOVE to CCR instruction. Note the size is 1, ie Word.

6280 Type 14 is the MOVE to SR instruction.

6310 Type 15 is the MOVEM register-to-memory instruction. It uses reglist and back to convert the following word into standard syntax register list.

6410 Type 16 is the MOVEM memory-to-register instruction.

6460 Type 17 instructions require an address followed by a data register.

6510 The only type 18 instruction is LEA, which needs an address followed by a data register.

6560 Type 19 instructions are the DBcc ops, and procedure do_cond is used to determine the condition, then the data register and branch destination are calculated.

6610 Type 20 instructions are the Scc ops requiring a condition and address.

6710 Type 21 instructions are the 'quick' forms of ADD and SUB, requiring immediate data, a register, and an address. The expression d=d+8*(d=0) is a fast way of setting d=8 if d was 0.

6910 Type 22 instructions are the branch on condition and BSR instructions. The condition is extracted from the opcode, and special action taken for BRA and BSR. Other conditions are calculated using procedure do_cond. Both short and long branch destinations are then worked out.

7060 The only type 24 instruction is MOVEQ, which requires data and a register number.

7086 Type 25 instructions are SCBD, SUBX,ABCD,

		and ADDX. They require either D?,D? or −(A?),−(A?).
7110		The only type 28 instruction is CMPM, which requires (A?)+,(A?)+.
7210		Type 29 instructions are CMP, CMPA and EOR. Note that these have empty initial values in the data table, and the initial part of the instruction is calculated in this part of the program.
7350		The only type 22 instruction is EXG, which needs to be followed by two registers. (Note that this one is not in 'order'.)
7510		Type 30 and 31 correspond to the instructions ADD, ADDA, SUB and SUBA.
7710		Type 32 instructions are the shift and rotate instructions. There is no initial value of p$ on entry to it; the whole opcode is constructed at this point.
8060–8998		The main data list for all the instructions. The numbers are expressed in decimal, which makes them harder to understand at a glance. Note that the final entry of 0,0,"??",0 ensures that any illegal instructions will get the opcode "??".
9000–9070	do_cond	Adds to p$ the condition code defined by N.
9100–9120	comp2	Gives a signed number for 8 bit quantities.
9130–9150	copm2_w	As comp2 but for 16 bit quantities.
9160–9230	do_dots	Adds the appropriate size mnemonic to the instruction.
9240–9300	bits	Adds the appropriate text for the 'bit test and ?' instructions.

To the best of my knowledge, the program disassembles correctly all the 68008 instructions, and calculates the destinations of all relative instructions. To use it, simply RUN, and there will be a pause while the array is set up. You can then enter the starting address and output stream, and if all is well you should see instructions you recognise. To test it, start disassembling at the switch-on location, defined by vector 0, and you should see quite an elaborate memory test, though you may not realise exactly what it does at first.

Converting it to machine code

This may seem a daunting prospect at first, but it's not really, as long as you take things easy to avoid silly mistakes. The complete conversion shouldn't take over 4K bytes, though on my version over 1K was devoted to data alone. With this sort of length of program, conversion without an assembler is a near impossibility, though a great speed improvement can be made by the addition of just 26 bytes of code, and I'll cover this first.

The slowest part of this program is undoubtedly the REPeat loop at line 1030. Replacing this by a small machine code routine speeds the whole program up by a factor of about 4, which is a substantial difference. What the machine code has to do is scan the list of opcodes, comparing them to the desired one, then return back to BASIC with a number corresponding to the value of 'i' at line 1040. As it is beyond the scope of this book to show how machine code can access BASIC arrays, the first two elements in the array oplist( ) have to be POKEd into memory, in the form of pairs of words. Let's have a look at the routine responsible for this great improvement.

Listing 9.2: Disassembler Improver

```
                    enters with D1=Op code
                    exits with number in RETPAR

41FA001A            LEA     TABLE(PC),A0
4280                CLR.L   D0                      zero count
5280        LOOP    ADDQ.L  #1,D0                   increase counter
3401                MOVE.W  D1,D2                   D2=op code
C458                AND.W   (A0)+,D2                mask bits
B458                CMP.W   (A0)+,D2                compare with op code
66F6                BNE     LOOP                    if not then try again
41FA000B            LEA     RETPAR(PC),A0
1080                MOVE.B  D0,(A0)                 store answer in RETPAR
4280                CLR.L   D0                      ready for BASIC
4E75                RTS                             and return
0000        RETPAR  DS 2                            space for result
....        TABLE   starting here a table of double word
                    entries, the first being the mask, the
                    second being the op code. The last entry
                    must be 0000,0000
```

It uses D0 as a counter, equivalent to 'i' in the SuperBASIC, and A0 as a pointer to the table. It scans the table until a match is found, then stores the value of D0 in location RETPAR.

At location TABLE you should POKE the words for the masks and opcodes, read from the data statements. (Note that the order of each pair has to be reversed before POKing.) Then, to use the routine, add some lines like

```
1030 CALL ????,op: REM start location
1040 LET i=PEEK(????+26)
```

Choose a sensible place for the routine — don't put it in the same place you put the routines you want disassembled, for example!

The full disassembler

Before starting conversion to a complete machine-code version, it is best to specify a few things. Firstly, I defined exactly how I was going to store the equivalent of the important variables, and in what registers.

The 68008 has so many registers that you don't have to store any variables (except 'P$') in memory — they can all be held in registers. The ones I chose were:

PC = A5
OP = D7 word
SIZE = D5 byte
'P$' = A4

The way I defined P$ was by having space of 40 bytes set aside for it, starting at location PSTR. At the start of the routine, I put the instruction

```
LEA PSTR(PC),A4
```

which set it to 'null string'. Then, every time something had to be added to it, I used auto-increment addressing. For example, if I wanted to add a bracket to it, I used

```
MOVE.B #"(",(A4)+
```

which stores the ASCII for '(' in the string, then increments A4 ready for the next 'addition' to it. When the instruction was complete, a

```
MOVE.B #10,(A4)
```

stored a line feed to store its end, ready for printing. If you do it this way, don't be tempted to do word and long operations directly on it. For example, if you had to add '(PC)' onto it, don't try

```
MOVE.L #"(PC)",(A4)+
```

as I did. When the program executes this, if A4 is not even, (there is a 50/50 chance it won't be) you will cause an address error, with nasty consequences.

The equivalents of the functions HEX$ and BHEX$ I wrote were very similar to the HEX4 and HEX8 routines in Chapter 4, but instead of calling PRINT, each ASCII byte was added by doing

MOVE.B D0,(A4)+

The BASIC program makes a lot of use of the SELect command, which has no real equivalent in 68008. To make up for this, I wrote a jump table calculator, a sort of machine code ON GOTO instruction. It wasn't as easy as I first thought, as both the routine and the data it used had to be position independent. The way around this was to make the table contain not simply a list of where each routine was, but a list of words defining how far away from the list the routines were. This can be done simply with an assembler by using the '*' operation, which stands for 'current value of the PC', so a list of entries would look something like this:

```
TABLE   DCW TYPE0-*
        DCW TYPE1-*
        DCW TYPE2-*
        DCW TYPE3-*
```

The DCW stands for 'define constant word', and may vary from assembler to assembler.

The routine that uses this data follows, and requires two parameters — the 'number' that the routine requires, starting at 0, in D0, and the location of the jump table, in A0.

Listing 9.3: Jump Table Calculator

```
                    On entry - D0.B=ITEM 0-127
                    A0=TABLE start

4880        TABCALC EXT.W   D0
48C0                EXT.L   D0                      D0=long
D080                ADD.L   D0,D0                   double D0
D1C0                ADDA.L  D0,A0                   add to table start
3010                MOVE.W  (A0),D0                 D0=table entry
48C0                EXT.L   D0                      make it long
D1C0                ADDA.L  D0,A0                   get absolute address
4ED0                JMP     (A0)                    go to the routine
```

Firstly, D0 is made a long word, then doubled. This is added to the start address of the table, so A0 points to the relevant word in it. Next the word is read into D0, then made long. This is so that negative displacements can be specified in the table, and the displacement added to the location in the table, giving the absolute location. Finally, the routine is jumped to.

I hope this has given you some ideas for your own conversion — don't be afraid of trying it. It can be done in steps, testing as you go. One possible improvement is testing for illegal addressing modes and sizes, for

the instructions that don't allow all the modes. This could be done with a data table for each of the allowed lists, and a parameter passed to the address calculator indicating which table is used for the instruction.

CHAPTER 10
Other 68000 Series Devices

The 68008 is just one of a series of processors from Motorola. The entire range is designed to be upwardly software compatible, though there are slight exceptions to this. The Sinclair QL is apparently the first of a series of machines from Sinclair to use the 68000 family, so we can expect their next machine to use another 68xxx processor.

In this chapter, we will look at the various other processors in the family, and some of their support chips. At the time of writing, some of these products exist on paper only — their specification is known, but the devices themselves may not be available.

Processors — from 8 bit to 32 bit

There are currently four processors in the series, summarised below:

Processor	*Features*
68000	16 bit data bus version, 24 address bits
68008	8 bit data bus, 20 address bits
68010	virtual memory version of 68000
68020	32 bit data and address bits, virtual memory processor

The 68000 — the flagship

The first in the series is the 68000. To the programmer it is practically identical to the 68008 used in the QL, except that code runs twice as fast as it would in the QL. The reason for the speed difference is because of the hardware. The 68000 has a 16 bit data bus, while the 68008 has only 8 bits. This means that all data and opcodes take half as long to read from memory, thus nearly doubling the comparative speed of operation. Another difference is that there are 24 address bits, so it can access up to 16 Mbytes of memory. In addition, all seven levels of interrupts are supported.

The 68008 — reduced version

This is the processor in the QL, and is a cut-down version of the 68000. It has only 8 data bits, and 20 address bits, thus addressing up to 1 Mbyte. It is designed for use in low-cost applications, and its narrow data bus is

easier (and cheaper) to interface to 8 bit peripheral devices. This was the primary consideration in its choice for the QL, as it reduced the number of pins necessary on the expensive custom chips.

The 68010 — virtual memory processor

This is software-compatible with the previous processors, with one exception, but its major advantage is that it supports virtual memory. What this means is that, to the programmer, it seems as if a large amount of the 16 Mbyte address space contains memory, where in actual fact there is a much smaller amount of RAM physically there. This is done by storing memory contents on some form of secondary storage, such as big capacity disk drives. When the processor tries to access a section of memory that is not physically present, the access is paused while the secondary storage is read, then the instruction completes.

To control the virtual memory facilities, there are some register differences on the 68010. In addition to the normal register set, there is a 'vector base register', and two 'alternate function code registers'. Certain instructions have been added to the set to manipulate these registers, based on the mnemonics MOVEC and MOVES, for 'move control' and 'move address space'. The vector base register is used to store the location of the vector table, which lies at $000000 in the 68000/8, so that it can be moved elsewhere in the memory map. The only software incompatibility is with the MOVE from SR instruction, which becomes privileged in the 68010. Exceptions are handled differently, as more data is put on the stack, from 4 to 29 words. Bus and address errors are handled in a different way, to aid virtual memory operation.

The final extra feature of the 68010 is its special 'loop-mode' operation. It allows small DBcc loops to execute very much faster than they would normally, by storing all the opcodes of the loop inside the processor. This cuts down on memory access drastically, and offers great speed improvements. It is particularly impressive when doing block move and search operations.

The 68020 — 32 bit virtual memory

The 68020, when available, will arguably be the world's most powerful microprocessor. It has all the features of the 68010, but with the addition of a full 32 bit data *and* address bus, as well as a fast co-processor interface. It is the first processor to have both 32 bit data and address bus, which means it can directly access up to a staggering 4096 Mbytes of memory. Its wide data bus makes it very fast indeed, as it can fetch 32 bits of data in a single cycle, while its co-processor interface gives it great expansion capabilities.

As well as the extra registers of the 68010, it has another A7 register, the master stack pointer, a 'cache control register' (CACR) and a 'cache

address register' (CAAR). Its instruction set is upward-compatible with the previous processors in the family, but has been greatly extended. As well as byte, word and long sizes of operations, quad word operations can be executed, which give 64 bit calculations. Four further addressing modes have been added, to improve its indirect memory operations, and the handling of BCD data has been improved with the instructions PACK and UNPCK. There are extra system traps, the DIV and MUL instructions operate on 32 bit parameters, and the Bcc and BSR instructions are extended on it to give 32 bit displacements. Two of the unused bits in the status register are used in the 68020, namely bit 14 as a second trace bit, and bit 12 as a master/interrupt flag.

The hardware of the 68020 includes many features for efficient multiprocessing, and has instructions to handle co-processors. In particular, it can work in conjunction with the 68881 maths processor, detailed later, producing a very powerful combination.

Support chips

There are several peripheral devices available for the 68000 series, some more specialised than others. They are summarised below:

Device	Description
68881	Floating point maths co-processor
68451	Memory management unit (MMU)
68450	DMA controller
68465	Floppy disk controller
68564	Serial I/O controller
68681	Dual UART
68486	Raster memory interface
68487	Raster memory controller

Most of these are standard, and can be found in most manufacturers' product lists. The MMU is a device for providing memory protection across the 16M range of the 68000, so that programs running in user mode can be prevented from accessing certain areas of memory.

A DMA controller is used for the fast transferring of data between memory and peripherals, by using the processor only for setting up the parameters for the transfer. The actual transferring does not require the processor's time.

A serial I/O and a UART perform similar functions, namely the transfer of data bit-by-bit to and from the processor. They differ in the protocols used during the transfer.

The 68881 is a floating point maths processor, with a very powerful specification. It was specially designed to co-process with the 68020, but

can be used as a peripheral by other processors, including the 68008, but at reduced performance. It has eight 80 bit data registers, a 32 bit control register, a 32 bit status register, and a program counter. It has its own instruction set, and uses four data types — single precision, double precision, extended precision and packed real decimal string. As each register has 80 bits, its maximum range is a number with 64 bits of mantissa, and 15 bits of exponent, giving a maximum value of 2↑32768, which I can't work out on any calculator. Its mantissa size means it can accurately hold a number like 12345678901234567, which is an awful lot. Its instruction set includes most operations you would never need to do with floating point numbers, as well as conversions and conditional jumps.

As well as the expected operations, such as +, −, * and /, it also does square root, trig functions, inverse trig functions, hyperbolic functions, logs and certain powers, and a few more besides! It does many of these operations not just on its own registers, but also directly on the 68020's memory, and it does its calculations, wherever possible, at the same time as the 68020 executes its own programs. It is *very* fast indeed — for example, it can do over 338,000 additions or 252,000 precision multiplications in a single second, easily outperforming most other methods.

The 68486/7 pair of devices are compatible with all the 68000 series, the earlier 6800 series, and can also be made to work with most other processor types. Together they offer a very flexible graphics system, but without requiring very much extra support electronics. They can address from between 16K and 1 Mbyte of video memory, which can be shared with the host processor, and can use the most popular types of RAM chips, both available and forthcoming. They are controlled by registers, and there are many different modes that can be selected. In bit-plane graphics mode, four colours can be displayed in 640 pixels, or 16 colours in 320 pixels, with the displayed colours selected from a palette of 4,096 different shades. List-mode graphics are primarily for character displays, and offer definable graphics but with fast screen alteration. (This is a disadvantage of the QL — characters displayed on the bit-mapped screen take many bytes and quite a bit of calculation, whereas character-mapped screens can use as little as one byte per character.) In 'true object' mode, up to eight sprites are available, which are independent shapes that have their own characteristics, along with collision detection and priority control.

Smooth scrolling in any direction is made easy and fast by the hardware, instead of the slow block-moves required on machines such as the QL. The 68486 chip handles the RAM, including refreshing and timing, and general control information. The 68487 converts the data in the memory and registers into video pixel information, which can then be

converted into signals for a monitor or television. Both the American and European signal standards are supported, which should make conversion between the two easier. It would be a relatively easy matter to interface these devices to the QL, and would make an interesting add-on.

Other 68000 series machines

There are few microcomputers available at the time of writing that use a 68000 series processor. The QL is the only one to date that uses the 68008, while there are a few machines that use the 68000, the two most popular currently being the Sage series, and the Apple Macintosh. The latter machine was used to write the text and illustrations for this book, and the quality of its firmware and supplied software shows just how powerful the 68000 can be when programmed well. As yet there are no commercially-available machines that use the 68010 or 68020, though rumours abound that there will be shortly. As they are upward-compatible, you can use the skills learnt on your QL's 68008 on any subsequent machines, so your knowledge is, to coin a phrase, 'future proof'.

EPILOGUE

Multi-tasking — an Example

Multi-tasking is quite an advanced subject but, just to give you a taster of it, there is a multi-tasking program that gives a trace function to SuperBASIC. It does this by setting up what is known as a job, which is an independent program, whose sole purpose is to print at the top of the screen line numbers, when they change.

To start the trace, CALL INIT, and this sets up the job, and copies it on to its allocated memory area. CALL PRIOR,xx sets the relative speed of the trace to 'xx'. Eight is usually sufficient, but 16 or even 32 may be required in some cases for best results.

I chose a window at the top of the screen above the usual windows for the output. If your television doesn't display that area, simply change the text that defines the window, before you call INIT. The listing is given in **Figure A**.

When trace is enabled, the RESPR function is not usable, giving a 'not complete' error, and a MODE instruction will disable the trace output. To temporarily switch off trace, set a speed of 0.

How it works

Multi-tasking is a major feature of QDOS, and can get quite involved. I shall give only an overview of it here.

To multi-task, a *job* has to be set up. A job is an independent program that is entirely self-contained, including its own stack and data areas. Normally on the QL, there is only one job, namely SuperBASIC, but the system can cater for up to 255 simultaneous jobs at any time. Each job also has its own channels and RAM exception vector tables. If at any time the job goes into supervisor mode, multi-tasking is disabled until user mode is re-entered.

Subroutine INIT configures the job, copies it to the proper place, and activates it. It does this by using two QDOS traps, and a DBF loop to copy the program. When setting up a job, a space of suitable size is allocated in memory, just under RESPR, and in this program a size of 100 bytes is chosen as the length. This is long enough for both the program and a suitable amount of stack space. After the trap, the space allocated is in A0, so the actual program is then copied into this area. Next, another trap

```
                    Setup job

7200      INIT      MOVEQ  #0,D1                 signal 'independant job'
243C0000            MOVE.L #100,D2               set length
0064
4283                CLR.L  D3                    set data to 0
2243                MOVEA.L D3,A1                and start address
7001                MOVEQ  #1,D0                 signal 'create job'
4E41                TRAP   #1                    create it
43FA002C            LEA    TRACEJOB(PC),A1       A1=start
243C0000            MOVE.L #LENGTH-1,D2          and length
004A
10D9      STCOPY    MOVE.B (A1)+,(A0)+           copy in into its area
51CAFFFC            DBF    D2,STCOPY
103C000A            MOVE.B #$0A,D0               signal 'activate'
7401                MOVEQ  #1,D2                 priority=1
7600                MOVEQ  #0,D3                 timeout=0
4E41                TRAP   #1                    activate the job
7208                MOVEQ  #8,D1                 speed=8

                    Set speed of trace to D1

22790002  PRIOR     MOVEA.L $28068,A1            A1=job table
3068
22690004            MOVE.L 4(A1),A1              A1=2nd job header
13410013            MOVE.B D1,13H(A1)            store new speed
4280                CLR.L  D0                    ready for BASIC
4E75                RTS                          and return

                    The Job itself

4FFA0062  TRACEJOB  LEA    TRACEJOB+100(PC),A7   set the stack
41FA0032            LEA    SCR(PC),A0            channel name
7001                MOVEQ  #1,D0                 signal 'open'
72FF                MOVEQ  #-1,D1                job ID
7602                MOVEQ  #2,D3                 new file
4E42                TRAP   #2                    open the channel
7EFF                MOVEQ  #-1,D7                set initial line number to -1
22790002  TLOOP     MOVEA.L $28010,A1            A1=BASIC area
3010
322900D0            MOVE.W $D0(A1),D1            current line number
3E41                CMP.W  D1,D7                 has it changed?
67F2                BEQ    TLOOP                 if not
3E01                MOVE.W D1,D7                 D7=latest no.
123C002D            MOVE.B #"-",D1
76FF                MOVEQ  #-1,D3                timeout
7005                MOVEQ  #5,D0                 signal 'output'
4E43                TRAP   #3                    print a "-"
3207                MOVE.W D7,D1                 current line no.
34790000            MOVEA.W $CE,A2               get the vector
00CE
4E92                JSR    (A2)                  print it in decimal
60DA                BRA    TLOOP                 and go round again
000F      SCR       DCW    15                    length of channel name
5343525F            DCB    "SCR_400X12A40X4"     channel name
34303058
31324134
305834
00
00000000            DB     7
0000
          LENGTH    EQU    *-TRACEJOB            length of job
```

Figure A: Trace Using Multi-tasking.

is done, which actually activates the new job, then control passes to PRIOR.

Subroutine PRIOR alters the relative speed of the job by altering the relevant byte in the job table, and putting the value of D1 into it.

TRACEJOB is the actual job, which starts by setting the stack pointer to the end of the allocated area. A channel is then opened, using TRAP #1, then D7 initialised to a value of −1. D7 is used as the 'last line number printer' parameter, so the number only gets printed when it changes.

At TLOOP, the current line number is read from the BASIC area, and tested to see if it has changed. If it has not, then control passes back to TLOOP until it has. When it does change, the new number is stored in D7, and a '–' sign printed, which is used as a separator between numbers. Next, the vector at location $00CE is called, which is a routine to 'print a decimal number', which requires the number in D1, and the channel ID in A0. The original value of A0 is never altered after the trap to open it, so its value doesn't need to be explicitly set.

At SCR, the output channel is specified, and this is done by having a word holding the name length, followed by the ASCII of the name. That chosen lies directly above the standard windows when in TV mode, but if you have a monitor, or your TV cannot display this position, alter it to suit. Don't forget to alter the word that defines the length, though.

References

M68000 Programmer's Reference Manual, Motorola Ltd.
68000 Assembly Language Programming, Kane, Hawkins & Leventhal.
QDOS Manual, Sinclair Research Ltd.

The Motorola book is *the* guide to the 68000/8. As it is written by its manufacturers, it contains all the technical information you should ever need, though its use for actual programming is limited. Be sure you get the edition that mentions the 68008.

The Leventhal et al book is full of programming examples and methods, though it is not for the raw beginner. The chapter on debugging is particularly good, and includes the well-tested advice 'Sometimes the following approach may be your best bet: turn off your computer, and have a beer.'

I used a preliminary version of the QDOS manual, so I'm not sure what the final version will be like. My version contains lots of information, but is difficult to understand.

Index

A

ABCD 74
ADD 33, 75
ADDA 33, 76
ADDI 33, 77
ADDQ 33, 77
ADDX 78
Addressing modes 13 ff, 69 ff
Alternate screen 67
AND 34, 79
ANDI 34, 80
ASL/R 81

B

Binary 2
Branches 26, 29, 83, 86
BSR 86
Bit test instructions 84

C

CALL command 20, 43
CHK 87
CLR 33, 88
CMP 28, 88
CMPA 29, 89
CMPI 29, 90
CMPM 91
Condition codes 8, 25

D

DBcc 31, 91
Disassembler 133 ff
DIVS 93
DIVU 94

E

EOR 35, 94
EORI 35, 95
Exceptions 47 ff
EXG 42, 96
EXT 97

H

Hexadecimal 2
Hex loader 11
Hex printing 38

I

ILLEGAL instruction 97
ILLEGAL exception 49
Interrupts 51
IPC 8049 66

J

Jump table calculator 153
JMP 27, 98
JSR 33, 99

K

Keyboard scanning 67

L

LEA 41, 99
LINK 100
LSL/R 101

M

Memory map 10
Memory of screen 58
MOVE 13 ff, 102
MOVEA 22, 104
MOVEM 22, 106
MOVEP 108
MOVEQ 22, 109
MULS 109
Multi-tasking 161
MULU 110

N
NBCD 111
NEG 43, 112
NEGX 112
NOP 42, 113
NOT 42, 113

O
OR 35, 114
ORI 35, 114

P
PEA 116
Plotting 62
Printing characters 64

Q
QDOS 44, 52, 65, 161
QDOS, disabling of 68

R
RAM exception vectors 54
Registers 8
Reset 49
Reset instruction 117
RESPR 9
ROL/R 117
ROXL/R 119
Rotates 35
RTE 120
RTR 121
RTS 32, 121

S
SBCD 122
Scrolling screen 32
Screen memory map 58
Set if condition (Scc) 122
Shifts 35
Sign extending 4
Stack pointer 21
Status registers 8
STOP 123
SUB 25, 124
SUBA 28, 125
SUBI 28, 126
SUBQ 28, 126
SUBX 127
Subroutines 32
Supervisor mode 9
SWAP 43, 128

T
TAS 128
Tracing, using T bit 50
Tracing BASIC programs 161
Traps 52
TRAP instruction 129
TRAPV 129
TST 130
Two's complementing 4

U
UNLK 130
User mode 9

V
Vector table 48

Assembly Language Programming on the Sinclair QL explains all you need to know about programming the 68008 microprocessor, one of the most powerful chips currently available. It also shows how to use the hardware and software facilities of the QL, including the 8049 second processor.

Starting from the basics, the internal structure and register set is explained, and the many addressing modes clearly explained. The use of the important traps and exception handlers is shown, as well as explaining the methods of adding your own.

Many useful and explanatory machine code listings are given, along with relevant BASIC procedures and functions, including a full disassembler. The most important features of QDOS are revealed, along with how best to use them in your own programs.

Machine code programming on powerful processors can be a daunting prospect using complex data books, but Assembly Language Programming can show you how easy and powerful it can be on the Sinclair QL.

Andrew Pennell is a university student and freelance programmer, and contributes to several magazines, including Popular Computing Weekly. He has written software for several companies, including Sinclair Research.

GB £ NET +007.95

ISBN 0-946408-42-4

00795

9 780946 408429

SUNSHINE

ISBN 0 946408 42 4

£7.95 net